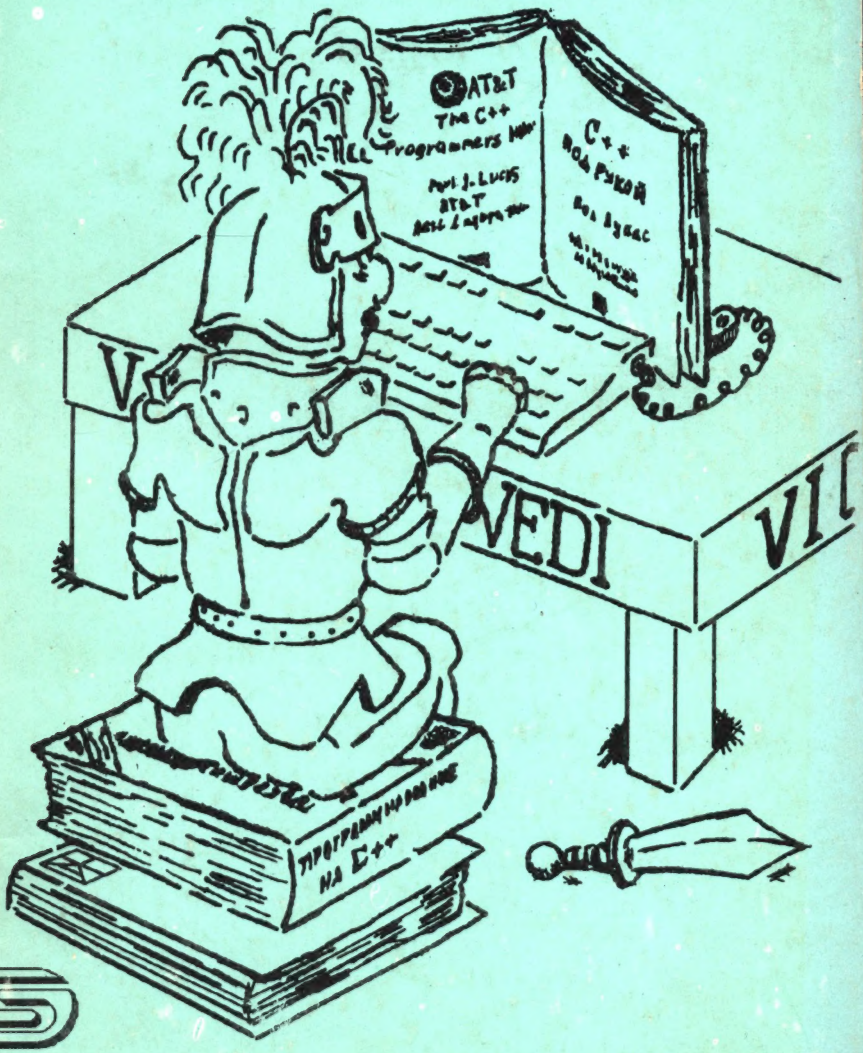
Пол Лукас

C++ под рукой

ЯЗЫКИ ПРОГРАММИРОВАНИЯ



C++
под рукой.

The C++ Programmer's HandBook

Paul J. Lucas

AT&T

Bell Laboratories

Prentice Hall, Englewood Cliffs, New Jersey 07632

C++
под рукой

Пол Лукас

Перевод с английского
К. В. Сулема

Под редакцией
С. Н. Сущенко

Киев НИПФ «ДиаСофт» 1993

ББК 32.973
С 45

Лукас П.

С 45

С++ под рукой: Пер. с англ. - Киев: «ДиаСофт», 1993. - 176 с., ил.

ISBN 5-87458-213-8

Книга американского специалиста представляет собой удобный справочник, позволяющий быстро найти нужную информацию по С++, препроцессору, а также по библиотеке ввода/вывода и другим библиотекам. В книге не описана какая-либо конкретная версия компилятора С++. Здесь дан стандарт языка С++ фирмы AT&T, в которой этот язык и был разработан Бьярном Страуструпом. Этот стандарт был реализован фирмой Borland в компиляторе Borland С++ версий 3.0 и 3.1.

Книга содержит большое количество примеров, которые служат хорошей иллюстрацией к использованию средств языка. Во многих случаях делается сравнение конструкций С++ с соответствующими конструкциями С, при необходимости указываются различия между этими языками.

Книга рассчитана на программистов разной квалификации, использующих язык С++.

ББК 32.973

ISBN 0-13-118233-1 (англ.)

Published by Prentice-Hall, Inc.
A Simon & Schuster Company
Englewood Cliffs, New Jersey 07632

ISBN 5-87458-213-8 (русск.)

Предисловие переводчика

Предлагаемая вашему вниманию книга является кратким справочником по одному из наиболее популярных сейчас языков программирования C++. Этот справочник рассчитан в первую очередь на программистов, хорошо знающих C++, однако с равным успехом его могут использовать и новички. Кроме того, справочник может быть полезен и для разработчиков, использующих C и желающих перейти на C++.

Каждая синтаксическая конструкция, описываемая в книге, сопровождается примерами, поясняющими ее семантику. Примеры подобраны таким образом, что служат хорошей иллюстрацией к использованию средств языка. Во многих случаях делается сравнение конструкций C++ с соответствующими конструкциями C, при необходимости указываются различия между этими языками.

Особенностью этой книги является то, что она не описывает какую-либо конкретную версию компилятора C++. Здесь дан стандарт языка C++ фирмы AT&T, в которой этот язык и был разработан Бьярном Страуструпом. Следует отметить, что этот стандарт был реализован фирмой Borland в компиляторе Borland C++ версий 3.0 и 3.1.

Немного об авторе. Пол Дж. Лукас является членом Технического Совета в International Switching Systems Division компании AT&T Bell Laboratories в Нейпервилле, штат Иллинойс. Кроме того, он проводил бета-тестирование C++ версии 3.0. Его профессиональные интересы охватывают области, связанные с языками программирования, компиляторами, разработкой пользовательских интерфейсов и

интеллектуальных инструментальных систем, облегчающих жизнь программисту.

Хочу выразить благодарность сотрудникам фирмы «ДиаСофт» Игорю Хижняку и Юрию Артеменко за помощь, оказанную при подготовке этого перевода.

Желаю больших успехов в творчестве!

К. Сулема

Киев, фирма «ДиаСофт»

24 мая 1993 г.

Предисловие

Эта книга была задумана как удобный справочник, позволяющий быстро найти нужную информацию по C++, препроцессору, а также по библиотеке ввода/вывода и другим библиотекам. Справочник может быть полезен как для новичков, так и для опытных программистов.

Для программиста на C++ эта книга должна стать другом, который сидит рядом с компьютером или терминалом, помогая найти нужную информацию и не перелопачивая при этом длинные параграфы.

Вся информация в книге имеет следующее представление:

Заголовок...

- Начинает новый предмет.

- Может использовать многоточие ("...") для формирования предложений, окончаниями которых являются пункты списка, следующего за заголовком (таким же образом, как и в этом примере).

Заголовок...

- Начинает новую тему в текущем предмете.

- Также может использовать многоточие для формирования предложений, окончаниями которых являются пункты списка, следующего за заголовком.

Эта книга не является учебником по программированию на C++, однако в ней представлена техника использования определенных возможностей языка, а также различные

тонкости их взаимоотношений.

Благодарности

Благодарю Бьярна Страуструпа за предоставление нам C++, а также за ответы на наши вопросы. Благодарю Джима Коплайна, Брэда Юхаса, Дэниса Мэнкла, Уоррена Монтгомери, Гриффа Смита, а также "посторонних наблюдателей" за их коментарии к черновому варианту этой книги. Благодарю Валери Монро, которая выполняла для меня бета-тестирование и предоставила мне предварительную копию C++ версии 3.0 для тестирования моих примеров для этой версии. Благодарю также Морриса Болски за его оригинальный "Справочник программиста на C", который явился отправной точкой для этой книги.

Нейпервиль, штат Иллинойс *Пол Дж. Лукас*

Библиография

AT&T Unix System Laboratories: *Unix System V-USL C++ Language System: Product Reference Manual, Release Notes and Seleced Readings*, releases 2.0-3.0 beta, Unix Press, 1989-1991.

Morris I. Bolsky: *The C Programmer's Handbook*, Prentice Hall, New Jersey, 1985.

Margaret A. Ellis and Bjarne Straustrup: *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, Massachusetts, 1989; second edition 1991.

Bjarne Straustrup: *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts, 1986; second edition 1991.

Bjarne Straustrup: *Parametrized Types for C++*, Journal of

Object Oriented Programming, Jan./Feb. 1988.

Язык C++

Язык программирования C++ базируется на своем предке - языке C. Хотя большая часть "подмножества C" в C++ не изменена, имеются некоторые отличия. Кроме того, с момента появления первой версии, C++ подвергся определенным изменениям. И те и другие отличия указаны в примечаниях.

Комментарии...

- Начинаются значком /*, заканчиваются значком */. Эти комментарии не могут быть вложенными.

- Начинаются значком //, заканчиваются символом новой строки*. Код содержащий эти комментарии может быть закомментирован при помощи /* и */.

```
/* Это один большой комментарий.
for( i = 0; i < 10; ++i ) // Инициализировать вектор
  a[ i ] = 0;
*/
```

- Допускают любое число пробелов.

Идентификаторы...

- Представляют собой последовательность букв, подчеркиваний или цифр, начинающуюся с буквы или подчер-

*Запрещается использовать комментарии // в строках препроцессора (см. "Комментарии" в разделе "Препроцессор").

кивания*.

 i, word_count, pi2

- Различают регистр, то есть foo - это не то же самое, что Foo.
- Могут иметь произвольную длину**.

Ключевые слова...

- Зарезервированы языком (выделены ключевые слова, появившиеся в C++):

asm	auto	break	case	**catch**
char	**class**	const	continue	**default**
delete	do	double	else	enum
extern	float	for	**friend**	goto
if	**inline**	int	long	new
operator	**overload**	private	**protected**	**public**
register	return	short	signed	sizeof
static	struct	switch	**template**	**this**
throw	try	typedef	union	unsigned
virtual	void	volatile	while	

Ключевые слова catch, throw и try используются для обработки исключительных ситуаций, которая пока в C++ не реализована.

Первые версии C++ используют ключевое слово overload, которое перестало быть необходимым. Кроме того, в этих версиях не реализованы ключевые слова asm, private, protected, signed, template и volatile.

---

* Не начинайте свои идентификаторы с подчеркивания или двойного подчеркивания (__). Такие идентификаторы компилятор C++ использует для своих нужд.

** Однако, некоторые реализации имеют ограничения.

В некоторых реализациях зарезервированы также ключевые слова fortran и pascal.

Ключевое слово asm не рассматривается в этом справочнике по причине специфической реализации.

Константы

Целые знаковые константы...

- Десятичные: Это необязательный знак, за которым следует ненулевая цифра, за которой, в свою очередь, следует произвольное число цифр.


```
23, -42, +1991 // + допустим, но необязателен
```

- Имеют тип int, если значение константы не превышает максимального допустимого значения для числа типа int на данной машине, в противном случае имеет тип long.

Целые беззнаковые константы...

- Это неотрицательные целые или длинные целые константы.

- Десятичные: Это целые или длинные целые константы без знака с добавленной в конце буквой U или u.

```
23U, 42u, 1991UL
```

- Восьмиричные: Это ноль, за которым следует произвольное количество цифр в диапазоне от 0 до 7.

```
027, 052, 03707
```

- Шестнадцатиричные: Это 0x или 0X, за которым следует произвольное количество цифр или букв в диапазоне от

а до f или от A до F.

0x17, 0x2a, 0X7C7

- Имеют тип unsigned int, если значение константы не превышает максимального допустимого значения для числа типа unsigned int на данной машине, в противном случае имеют тип unsigned long.

Длинные целые константы...

- Это знаковые или безнаковые целые константы, которые имеют большие значения, нежели это допустимо для чисел типа int.

- Явно обозначаются добавлением символа L или l к знаковой или беззнаковой целой константе.

23L, 0x2aL, 1991ul

Константы с плавающей точкой...

- Имеет целую часть, десятичную точку, дробную часть, символ e или E и экспоненту со знаком или без него.

- Могут не иметь целой или дробной части, а также не иметь десятичной точки или экспоненты.

3.14159, 3E8, 1., .2, 1.602e-19

- Имеют тип double, за исключением случаев, когда оканчиваются символом F или f (тип float) или символом L или l (тип long double).

```
23F, 42f, 1991F                 // float
3.14159L, 3E8L, 1.602e-191  // long double
```

Символьные константы...

- Чаще всего представляют собой отдельный символ, заключенный в одиночные кавычки. Имеют тип char*.

'm', '7', '+'

- Имеют числовые значение, которые равны значениям машинных кодов соответствующих символов.

'A' в кодировке ASCII имеет значение 65.

- Используют управляющие последовательности для представления определенных символов:

Ноль	NUL	'\0'
Тревога (звонок)	BEL	'\a'
Новая строка	NL (LF)	'\n'
Горизонтальная табуляция	HT	'\t'
Вертикальная табуляция	VT	'\v'
Забой	BS	'\b'
Возврат каретки	CR	'\r'
Прогон формата	FF	'\f'
Обратная косая	\	'\\'
Одиночная кавычка	'	'\''
Битовая маска**	0ddd	'\ddd'
Битовая маска	0xddd	'\xddd'

- Могут также представлять собой множество символов, заключенное в одиночные кавычки. В этом случае имеют тип int, но фактическое значение зависит от реализации.

'ah', 'MPNT'

---

*В С символьные константы имеют тип int.

**Допускаются битовые маски, которые могут быть заданы последовательностью шестнадцатиричных или восмиричных цифр, содержащей от одной до трех таких цифр.

Строковые константы...

- Это ноль и более символов, окруженных двойными кавычками.

```
"m", "C++", "No way, Jose!"
```

- Имеют тип static char, то есть это статический вектор символов.

- Содержат в конце добавляемый компилятором символ /0.

```
charfive[] = "Five"; // five содержит 5 элементов
```

- Используют \" для включения кавычек в строку.

- Могут быть записаны в нескольких строках. Последним символом в переносимой строке должен быть символ \. При этом символы \ и новой строки отбрасываются.

```
char    *quote = "/"Diplomacy/" is letting them /
have it your way."
```

- Соединяются, если являются смежными*.

```
// Ниже приведены две одинаковые строки
greeting1 = "Hello world!";
greeting2 = "Hello " "world!";
```

Перечислимые константы...

- Это идентификаторы,

---

*Эта возможность доступна при использовании препроцессора.

- Это единственный интегральный тип* (см. "Перечисления" в разделе "Объявления").

Выражения...

- Это сочетание одной или более констант, переменных, или вызовов функций и нуля или более операций.

```
i = 0, C++, c = sqrt( a*a + b*b )
```

- Рассматриваются как true, если результат их вычисления не равен нулю, или как false, если результат равен нулю.

Операции

При описании операций будут использовться следующие сокращения:

e	Произвольное выражение.
v	Любое выражение, которое может принимать значение.
i	Целое или символ.
a	Арифметическое (целое, символ или с плавающей точкой).
p	Указатель.
s	Структура, объединение или класс.
m	Член структуры, объединения или класса.
f	Функция.

Эти сокращения могут комбинироваться. Например, ie - это выражение целого типа.

*В C считается, что перечисления имеют тип int.

Арифметические операции

ae + *ae*	Сумма.
pe + *ie*	Адрес *pe* + *ie* * sizeof(**pe*)[*].
ae - *ae*	Разность.
pe - *ie*	Адрес *pe* - *ie* * sizeof(**pe*).
pe - *pe*	Число элементов вектора, которые имеют тип **pe*[**].
-*ae*	Минус *ae*.
ae * *ae*	Произведение.
ae / *ae*	Частное.
ie % *ie*	Остаток от деления[***].

Логические операции

!*e*	Отрицание.
e1 || *e2*	Или. *e2* вычисляется только, если *e1* имеет значение false.
e1 && *e2*	И. *e2* вычисляется только, если *e1* имеет значение true.

```
if( p && !*p ) // если p == 0, то !*p не вычисляется
   *p = 1;
```

Операции отношения...

- Выбирают одно из арифметических или адресных выражений[****].

---

[*] Если *pe* указывает, скажем, на массив double, то *pe* + 1 указывает на следующий элемент этого массива, а не на следующий байт в памяти.

[**] Это имеет смысл только, если оба указателя указывают на один и тот же вектор. Результат имеет тип ptrdiff_t, который определен в файле <stddef.h>.

[***] Знак результата не определен, если один из операндов или оба операнда имеют отрицательное значение.

[****] Для адресных выражений операции <, <=, > и >= имеют смысл только, если оба выражения указывают на один и тот же вектор.

==	Равно.
!=	Не равно.
<	Меньше.
<=	Меньше или равно.
>	Больше.
>=	Больше или равно.

```
inline int Inrange( int n, int lower, int upper )
{
  return lower <= n && n <= upper;
}
```

Операции присваивания...

v = *e*	Присваивает *v* значение *e*.

- Допускают множественное присваивание, выполняемое в одной операции. При этом присваивания выполняются справа налево.

```
a = b = c = 0; // a = (b = (c = 0));
```

- Могут сочетаться с арифметическими или битовыми логическими операциями. Например,

v операция = *e*

является сокращенной записью

v = *v* операция (*e*)

причем v вычисляется только один раз.

```
i += j, x <<= 1
```

Операции инкремента и декремента

iv++	Увеличить на *iv* 1; результатом выражения является значение *iv* до увеличения.
++*iv*	Увеличить на *iv* 1; результатом выражения является значение *iv* после увеличения.
pv++	Увеличить на pv на значение sizeof(**pv*); результатом выражения является значение *pv* до увеличения.
++*pv*	Увеличить на pv на значение sizeof(**pv*); результатом выражения является значение pv после увеличения.
--	Имеет ту же семантику, что и ++ но вызывает уменьшение на 1.

```
int i = 3, j, k;
j = i++;      // i = 4, j = 3
k = --j;      // j = 2, k = 2

char buf[3], *p = buf;
*p++ = 'a';      // buf[0] = 'a', p = &buf[1]
*++p = 'c';      // p = &buf[2], buf[2] = 'c'
```

Операции указателей и массивов

&*v*	Адрес *v*.
**pe*	Содержимое переменной, адресуемой посредством *pe*.
**fpe*	Функция, адресуемая посредством *pe* (см. "Вызов" в разделе "Функция").
pe[*ie*]	Доступ к элементу *ie* в массиве *pe*. Это сокращенная запись для **(pe + ie)*.

```
// Эти две функции функионально эквивалентны
void VecCopy( char to[ ] , const char from[ ])
{
  for( int i = 0; to[ i ] = from[ i ]; ++i );
}

// Эта функция, однако, имеет более краткую запись
void PtrCopy( char * to, char * from )
{
  while( *to++ = *from++ );
}
```

Операции структуры, объединения и класса

sv.m	Поле структуры, объединения или класса.
spe->m	Сокращенная запись для (**spe*).*m*.
sv.mf	Функция - член структуры, объединения или класса.
spe->mf	Сокращенная запись для (**spe*).*mf*.
*sv.*mp*	Поле структуры, объединения или класса.
*spe->*mp*	Сокращенная запись для (**spe*).**mp*.
*sv.*mfp*	Функция - член структуры, объединения или класса.
*spe->*mfp*	Сокращенная запись для (**spe*).**mfp*.
s::m или *s::mf*	Определяет *m* или *mf* как член структуры, объединения или класса.

```
struct Point { int x, y; };
Point a, b, *pb = &b;
a.x = a.y = pb->x = pb->y = 1;

int Point::*coord = &Point::x;
```

```
a.*coord = pb->*coord = 2;        // a.x = b.x =2
```

Побитовые операции

~ie	Дополнение до единицы.
ie1 << *ie2*	Сдвиг *ie1* влево на *ie2* бит. В освобождающиеся биты записываются нули.
ie1 << *ie2*	Сдвиг *ie1* влево на *ie2* бит. В освобождающиеся биты записываются нули, если *ie1* имеет беззнаковый тип, либо неопределенное значение, если *ie1* имеет знаковый тип.
ie1 & *ie2*	Побитовое И.
ie1 | *ie2*	Побитовое ИЛИ.
ie1 ^ *ie2*	Побитовое исключающее ИЛИ.

```
char letterA = 'A';  // Двоичное: 01000001
char letterB = 'b';                 // 01100010
const char mask = 0x20;             // 00100000
char lower = letterA | 0x20;        // 01100001
char upper = letterB | ~0x20;       // 01000010
```

Прочие операции

e1 ? *e2* : *e3* Если выполняется условие *e1*, то вычисляется *e2*, и результатом операции является *e2*. В противном случае вычисляется *e3*, и результатом операции является *e3*.

```
inline int Max( int a, int b ) { return a > b ? a : b; }
```

e1 , *e2* Вычисляется *e1*, затем *e2*. Результатом операции является *e2*.

```
void Reverse( int a[], int n )
{
```

```
  for( int i = 0, j = n-1;  i < n;  ++i, --j )
  {
    int t = a[ i ];
    a[ i ] = a[ j ];
    a[ j ] = t;
  }
}
```

::v	Доступ к глобальной переменной *v*.

```
int count;                  // Глобальная переменная
//...
void f()
{
  int count = ::count;  // Установить локальную в
                        //  глобальную
   //...
}
```

sizeof *e*	Число байтов байтов для типа выражения *e**.
sizeof(*type*)	Число байтов байтов для типа *type*.

```
char buf[ 100 ];
while( cin.getline( buf, sizeof buf - 1 ))
 cout << buf;
```

fe(e, e, ...)	Вызывает функцию с аргументами (см. "Вызов" в разделе "Функции").
(*type*)*e*	Значение *e* преобразуется к типу *type***.

* Выражение, содержащее операцию sizeof может быть вычислено на этапе компиляции.

** На жаргоне C и C++ эта операция называется "приведение типа" или просто "приведение".

type(e)	Значение *e* преобразуется к типу *type**.

```
char c, *cp = &c;
long cheat = ( long ) cp;  // Получить машинный адрес
```

Операции распределения памяти (только в C++)

new *type*	Выделяет пространство для типа *type* и возвращает адрес**. Для классов, кроме того, вызывается конструктор. При отсутствии достаточного объема памяти возвращает ноль.
new *type*[*ie*]	Выделяет пространство для массива из *ie* элементов типа *type* и возвращает адрес. Для классов конструктор вызывается для каждого элемента.
delete *pe*	Освобождает память, на которую указывает *pe****. Для классов вызывается деструктор. Если *pe* имеет нулевое значение, то операция delete не выполняет никаких действий.
delete[ ] *pe*	Освобождает память занятую массивом, на который указывает *pe*****. Для классов деструктор вызывается для каждого

*Это альтернативная запись приведения типа, которая введена в C++. Однако, она может использоваться лишь для "простых" типов, то есть для типов, которые не являются указателями * или ссылками &. Например, double(x) может быть использовано для преобразования x в тип double, а int *(p) не может быть использовано для преобразования p в указатель на int; вместо этого нужно использовать (int *)p.

**Соответствует функции malloc() в C.

***Соответствует функции malloc() в C.

****В первых версиях C++ между скобками необходимо было указывать число элементов массива.

элемента массива.

Существует глобальная переменная _new_handler, которая является указателем на функцию, не принимающую аргументов и возвращающую тип void. Как и все глобальные переменные, она инициализируется нулем.

Всякий раз, когда операция new не может выделить память, она проверяет _new_handler. Если значение _new_handler равно нулю, то возвращается ноль, в противном случае вызывается функция, на которую указывает _new_handler.

Обычно _new_handler указывает на функцию, которая сообщает пользователю о нехватке памяти и выполняет очистку (например, сохраняет файл на диск). Это позволяет не производить дополнительную проверку значения, возвращаемого операцией new.

```
void MyNewHandler()
{
  cerr << "Out of memory, sorry.../n";
  // ... здесь выполнить действия по очистке ...
   exit( 1 );  // Ненулевой код возврата*.
}
_new_handler = &MyNewHandler;
```

Приоритет и порядок выполнения операций

В таблице, приводимой ниже, операции расположены в порядке убывания приоритета. Операции с одинаковым приоритетом объединены в группы. Операции, выделенные курсивом, являются правоассоциируемыми, осталь-

*См. "exit()" в пункте "stdlib.h" раздела "Другие библиотеки".

ные - левоассоциируемыми.

::	*Доступ к глобальной переменной*
::	Обозначение члена структуры
()	Вызов функции
[ ]	Индексация
. ->	Выбор элемента
sizeof	*Размер*
!	*Логическое НЕ*
~	*Дополнение до единицы*
- +	*Унарный минус, унарный плюс*
++ --	*Инкремент, декремент*
& *	*Вычисление адреса, снятие ссылки*
()	*Приведение типа*
new delete	Выделение, освобождение
.* ->*	Выбор поля через указатель
* / %	Умножение, деление, модуль (остаток)
+ -	Сложение, вычитание
<< >>	Левый сдвиг, правый сдвиг
< <=	Меньше, меньше или равно
> >=	Больше, больше или равно
== !=	Равно, не равно
&	Побитовое И
^	Побитовое исключающее ИЛИ
|	Побитовое ИЛИ
&&	Логическое И
||	Логическое ИЛИ
? :	*Арифметическое если*
=	*Присваивание*
*= /= %= +=	
-= <<= >>=	
&= ^= |=	
,	Запятая

Порядок вычислений...

- Главным образом языком не определен.

```
n = ( a = x++ ) * ( b = x++ );
```

Пусть x имеет значение 1. Если операнды операции * вычисляются слева направо, то a будет равно 1, а b - 2. Если операция * является правоассоциативной, то b будет равно 1, а a - 2.

- *Всегда* имеет место левоассоциативность для четырех операций: &&, ||, ? : и , (запятая).

Арифметические преобразования

- Если один из операндов имеет тип long double, double или float, то другие операнды преобразуются к одному и тому же типу, причем long double имеет максимальный приоритет*.

- Иначе, если значения типа char, unsigned char, short int, unsigned short int или enum могут быть представлены типом int, то все операнды указанных типов преобразуются в int, иначе в unsigned int**.

- Затем, если один из операндов имеет тип unsigned long, то остальные преобразуются в unsigned long.

- Иначе, если один операнд имее тип long int, а остальные - unsigned int, и long int может представлять все значения unsigned int, то unsigned int преобразуется в long int, иначе оба операнда преобразуются в тип unsigned long

*В C все типы float автоматически поддерживают тип double, в C++ это правило больше не выполняется.

**В C все знаковые операнды преобразуются в int, а беззнаковые в unsigned int.

int*.

- Иначе, если один из операндов long или int, то остальные преобразуются к тому же типу, причем long имеет максимальный приоритет.

- Иначе, оба операнда имеют тип int.

Операторы

Оператор выражения...

- Представляет собой выражение, заканчивающееся точкой с запятой (;).

```
language = C++;
```

Оператор метки...

- Это оператор выражения, перед которым стоит идентификатор и точка с запятой**.

- Может быть использован в операторе goto (см. "goto").

```
OUTTA_HERE: return result;
```

Пустой оператор...

- Это просто точка с запятой.

- Используется, когда после метки или операторов while, do или for не должно следовать никакого оператора.

*В C все операнды преобразуются в unsigned long int.

**Идентификатор метки отличается от всех остальных идентификаторов тем, что его область действия ограничивается функцией, в которой он используется.

```
while( *p++ = *q++ );
```

Составной оператор...

- Представляет собой ноль или более операторов, заключенных в фигурные скобки.
- Допустим везде, где допустим оператор выражения.
- Определяет новую область действия, то есть переменные, определенные внутри составного оператора, являются локальными в нем и скрывают внешние переменные с теми же именами.

```
{ int t = a; a = b; b = t; }
```

- Имеет конструктор для локальных переменных класса, который вызывается при выходе из блока* (см. "Деструкторы" в разделе "Классы").

Оператор объявления...

- Описывает идентификатор и его тип** (см. "Объявления").

break;

- Прерывает выполнение последнего открытого оператора while, do, for или switch.
- Выполнение передается на оператор, следующий за прерваным.

---

*Кроме того, деструктор вызывается при выполнении операторов goto, break, continue и return.

**В C все объявления должны предшествовать всем операторам выражений, в C++ это ограничение снято.

```
for( int i = 0; i < 10; ++i )
  if( guess == a[ i ])
  {
    cout << "Congratulations!/n";
    break;            // Прерывает for
  }
if( i == 10 )    // Управление передается сюда
  cout << "Sorry.../n";
```

continue;

- Передает управление на начало последнего открытого оператора while, do или for. Эквивалентно оператору goto bottom, показанному ниже:

```
while( ... )
{
  // ...
  goto bottom;
  // ...
bottom: ;
}

do
{
  // ...
  goto bottom;
  // ...
bottom: ;
}while( ... );

for( ... )
{
  // ...
```

```
  goto bottom;
  // ...
bottom: ;
}
```

```
for( int i = 0; i < 10; ++i )
{
  if( a[ i ] == 0 )
  continue;// Ничего не делать иначе
  // ... выполнить код размещенный здесь ...
}
```

return выражение;

- Прерывает выполнение текущей функции и возвращает значение *выражения* вызвавшей функции. *Выражение* опускается, если функция имеет тип возвращаемого значения void.

- Преобразует тип *выражения* к типу возвращаемого функцией значения, если это необходимо.

goto метка;

- Передает управление оператору, помеченному *меткой*. (*Метка* должна находится внутри текущей функции)

```
goto FAIL;
```

- Не может вызывать передачу управления на оператор объявления с инициализацией, который расположен в том же блоке.

- Не может передавать управление в составной оператор, расположенный после оператора объявления с

инициализацией*.

- Может передавать управление за пределы составного оператора.

```
void Contrived()
{
  goto L1;   // ошибка...
  int a = 0;  // ... нельзя делать переход за этот
              // оператор
L1:
  goto L2;    // правильно...
  int b;      // ...нет инициализации
L2:
  goto L3;   // Правильно, но сомнительно
  goto L4    // ошибка...
  for( ; ; )
  {
    int c;
  L3:
    int d = 0;  // ... нельзя делать переход за этот
                // оператор
  L4:
    goto L5;  // Можно покидать составной оператор
  }
L5:
  ;
}
```

if(выражение) оператор1 else оператор2

- Вычисляет *выражение* и, если оно имеет ненулевой результат (true), выполняется *оператор1*, в противном

*Однако это допустимо в C.

случае - *оператор2*[*].

- Конструкция else и *оператор2* могут быть опущены.

Оператор switch

```
switch( выражение ) // первая форма (редкая)
  case константное-выражение1:
  case константное-выражение2:
  ...
    оператор
switch( выражение ) // вторая форма (общая)
{
  case константное-выражение1:
  case константное-выражение2:
  ...
    операторы
  case константное-выражение3:
  case константное-выражение4:
  ...
    операторы
  default:
    операторы
}
```

- Первая форма оператора switch (используемая редко) эквивалентна записи...

```
if( выражение == константное-выражение1 ||
    выражение == константное-выражение2 ... )
  оператор
```

**Оператор* не может быть оператором объявления, хотя допустимо использование составных операторов, которые содержат операторы объявления.

...за исключением того, что *выражение* вычисляется только один раз.

```
switch( n )
 case 2:  case 4:  case 6:  case 8:
   appreciate = n;
```

- Во второй форме вычисляется *выражение* и выполняется набор *операторов*, которые следуют за соответствующим *константным-выражением*. В противном случае выполняются *операторы*, которые следуют за default*.

- Заметьте, что выполнение оператора switch не заканчивается в конце пункта case, что позволяет выполнять множественные пункты case. В большинстве случаев для прерывания выполнения оператора switch используется оператор break.

```
switch( c )
{
   case '+':   n = a + b;  break;
   case '-':   n = a - b;  break;
   case '*':   n = a * b;  break;
   case '/':   n = a / b;  break;
   default:
     Error( "Bad operator: ", c );
}
//  Сюда будет передано управление после
//  выполнения  любого из операторов break
```

- Не допускается совпадение *константных-выражений*.

*Пункт default является необязательным, кроме того, он не обязательно должен быть последним.

while(выражение) оператор

- Представляет сокращенную запись для...

```
label:  if( выражение )
        {
          оператор
          goto label;
        }
```

do оператор while(выражение);

- Представляет сокращенную запись для...

```
label:  оператор
if( выражение ) goto label;
```

Оператор for

```
for( оператор1 ; выражение1 ; выражение2 )
  оператор2 *
```

- Представляет сокращенную запись для...

```
оператор1
while( выражение )
{
  оператор2
  выражение2;
}
```

...за исключением того, что оператор continue в *операторе2* будет вызывать продолжение процесса выполнения, начиная с *выражения2*.

---

* *Оператор1* - это *оператор*, то есть он заканчивается точкой с запятой.

- Если *оператор1* - это объявление*, то область действия объявленных переменных простирается до конца составного *оператора2*.

- *Оператор1* и оба *выражения* могут быть опущены (но точки с запятыми должны оставаться). Когда *выражение1* опущено, его значение считается ненулевым.

*В языке C *оператор1* может быть только выражением.

Организация программы

Программа на C++ обычно состоит из двух типов файлов: файлов заголовков и файлов кода. Файлы заголовков имеют расширение .h, а файлы кода - расширение .c. Достаточно сложная программа использует различные файлы каждого из этих типов.

Файлы заголовков содержат классы, шаблоны, структуры, объединения, перечисления и объявления функций, описания typedef, определения констант, функции inline и директивы препроцессора. Все вместе, это называется *интерфейсом* к файлам кода. Файлы кода содержат *реализацию* вашей программы.

Объявления...

- Описывают новый тип, переменную и ее тип, константу и ее значение или функцию, ее аргументы и возвращаемое значение.
- Присваивают тип int, если тип не определен явно.

Статические и автоматические классы памяти

Все переменные, определенные вне функций, относятся к *статическому классу памяти*, то есть они существуют все время, пока выполняется *программа**. Все переменные, определенные внутри функций, по умолчанию относятся к

*Примечание: Эти переменные также называют "глобальными".

автоматическому классу памяти, то есть они существуют только во время выполнения данной *функции*.

Чтобы переменная внутри функции относилась к статическому классу памяти, в ее объявлении нужно использовать префикс static, который является ключевым словом*.

```
void Dryer( int clothes )
{
  staticcling; // Существует столько же сколько и
               // программа
  // ...
}
```

По умолчанию статические переменные инициализируются значением ноль (или эквивалентным) до их использования и до инициализации на этапе выполнения (той, которая включает выражения). Автоматические переменные по умолчанию не инициализируются.

Статическая и внешняя область действия

Переменные, относящиеся к статическому классу памяти, объявленные вне функций, обычно имеют внешнюю область действия; то есть переменные, объявленные в одном файле, могут быть доступны в другом файле, если они объявлены с префиксом extern, который является ключевым словом.

```
// Файл 1
int global;
```

*Для явного объявления автоматических переменных используется ключевое слово auto. Это делают исключительно в целях улучшения читабельности программы.

```
// Файл 2
extern int global; // Доступ к global из 1-го файла
```

Чтобы сделать переменную, объявленную вне функции и относящуюся к статическому классу памяти, недоступной из другого файла, ее нужно объявить с использованием префикса static*. Ключевое слово static ограничивает область видимости переменной границами файла, в котором она объявлена**.

```
// Файл 1
static electricity; // Локальная в файле 1

// Файл 2
extern electricity; // Ошибка при связывании

// Файл 2a
int electricity; // Правильно
```

Область действия electricity из файла 1 ограничивается границами этого файла. Другая переменная electricity может быть объявлена в любом другом файле (файл 2а).

Простые типы...

- Это один из следующих типов:

char	Символ (один байт).
unsigned char	Беззнаковый символ.

*Функции также могут быть объявлены static с той же целью. Однако тот же эффект может быть достигнут, если прототип функции не включать в файл заголовка, поскольку в C++ все функции должны быть явно объявлены. Эта возможность включена только для совместимости с C.

**В этом контексте значение ключевого слова static изменяется с "постоянный" на "собственный".

signed char	Символ со знаком.
int	Целое (обычно слово).
unsigned int	Неотрицательное целое (имеет размер int).
short int	Короткое целое (слово или полуслово).
unsigned short int	Неотрицательное короткое целое.
long int	Длинное целое (слово или двойное слово)*.
unsigned short int	Неотрицательное длинное целое.
float	Число с плавающей точкой одинарной точности.
double	Число с плавающей точкой двойной точности.
long double	Число с плавающей точкой высокой точности.
void	Не имеет значения**.

Объявления unsigned, long и short эквивалентны объявлениям unsigned int, long int и short int соответственно. Является ли обявление char синонимом unsigned char или signed char, зависит от реализации компилятора.

Все это для целых типов гарантирует, что...

sizeof(short) ≤ sizeof(int) ≤ sizeof(long)

...хотя обычно long в два раза длиннее, чем short. Для типов с плавающей запятой гарантируется, что...

sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)

- Могут быть инициализированы при помощи выражения.

*Некоторые реализации могут иметь тип long long, который используется для представления очень больших целых чисел (учетверенное слово).

**Нельзя объявить объект типа void. Однако можно объявить указатель на void (см. "Указатели"). Кроме того, функция может иметь возвращаемое значение типа void, что указывает на то, что функция не возвращает никакого значения.

```
char broiled;
unsigned check = 197;
long distance = AT&T;
```

Перечисления...

- Позволяют использовать имена вместо числовых значений.

- Могут задавать имя типа.

- Могут быть инициализированы определенными (целыми) значениями. Может также быть установлено начальное значение, от которого начинается отсчет значений (по умолчанию отсчет начинается с нуля). Несколько элементов перечисления могут иметь одно и то же значение.

```
enum{ chocolate, vanilla, strawberry };
// chocolate = 0, vanilla = 1, strawberry = 2

enum Fruit{ orange, cherry, banana }; // Тип

enum Cost{ perPerson =2, perCouple /* = 3 */ };

Fruit florida = orange; // Обявление переменной типа
                        //  Fruit*
```

- Не могут дублироваться в пределах одной области видимости.

---

*Для объявления переменной перечисления в C ключевое слово enum должно использоваться явно (если только не используется typedef). В C++ это необязательно, но допустимо.

```
enum State { start, run, stop };
enum Position { start, middle, end }; // Ошибка
```

- Могут быть объявлены внутри классов (см. "Классы"). Область видимости таких перечислений ограничена классом, в котором они объявлены*.

```
class IceCream
{
  // ...
  public:
  enum Flavor { chocolate, vanilla, strawberry };
  Flavor  flavor;
  // ...
};
// ...

Flavor  favorite;  // Ошибка; Flavor отсюда недоступен
IceCream homeMade;
homeMade.flavor = chocolate; // Ошибка
homeMade.flavor = IceCream::chocolate; // Правильно
```

Векторы...

- Это непрерывные блоки памяти, хранящие множество элементов одного и того же типа.

- Объявляются посредством указания числа элементов, которое должно быть положительным целым константным выражением, заключенным в квадратные скобки [ ].

- Имеют только одну размерность. Многомерные массивы, используемые в других языках, в C++ представляются

*В ранних версиях C++ перечисления в классах не являются локальными.

вектором указателей на векторы.

```
float  vf[10];        // Вектор из 10 float
int    vvi[50][8];    // 50 векторов из 8 int каждый
char *vpc[5];     // Вектор из 5 указателей на char
```

- Могут быть инициализированы (если имеют статический класс памяти) набором значений, заключенным в фигурные скобки. Особый случай - это вектор символов (char), который может быть инициализирован строковой константой.

  Первое максимальное число элементов может быть опущено, оно равно числу инициализирующих значений. Если число значений меньше заданного числа элементов, то оставшиеся элементы инициализируются нулями (или эквивалентным значением). Если заданы все значения, то фигурные скобки могут быть опущены.

```
char greeting[ ] = "Yes, master?";  // 13 элементов*

int a[5] = { 7, 6, 9 }; // a[3] = a[4] = 0

int b[ ][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };  // [3][2]

int c[ ][2] = { 1, 2, 3, 4, 5, 6 };    // { } опущены
int d[ ][2] = { { 1 }, { 2 }, { 3 } };  // { } не могут быть
                                        // опущены
// d[0][1] = d[1][1] = d[2][1] = 0
```

Имя вектора само по себе является адресом первого эле-

*Не забывайте о том, что компилятор автоматически помещает нуль-терминатор в конец строки.

мента этого вектора:

$$v = \&v[0]$$

Единственным исключением является использование имени вектора в операции sizeof. При этом результатом операции будет число байт, занимаемых всем вектором.

Указатели...

- Это переменные, которые содержат адрес другой переменной или функции (см. также "Указатели на члены" в разделе "Классы").

- Объявляются при помощи символа *.

```
int i, *ip = &i;   // Указатель на int
int *ppi = &pi;  // Указатель на указатель на int
char (*pvc)[5];  // Указатель на вектор char[5]
float (*pf)(int); // Указатель на функцию принимающую
                  // параметр int и возвращающую float
```

- Предоставляют функциям механизм изменения значений аргументов*.

```
inline void Swap( int *a, int *b )
{
  int t = *a;
  *a = *b;
  *b = t;
}
// ...
Swap( &x, &y );  // Должны передаваться адреса
```

*В C это единственный путь изменения значения аргументов, в C++ возможно использование так называемых ссылок (см. "Ссылки").

- Могут иметь тип "указатель на void", который означает, что данный указатель адресует любой объект, не имеющий ни тип const ни тип volatile. Преобразование указателя в указатель на void должно быть явным. Для обратного преобразования указателя на void в указатель на реальный объект требуется операция приведения типа*.

```
char c, *p = &c;
void  *v = p;           // Явное преобразование
char  *q = (char *)v; // Необходимо явное приведение
```

Ссылки...

- Это второе имя для другой переменной.

- Объявляются при помощи символа *&*.

- При объявлении должны быть инициализированы. (Ссылка может быть инициализирована лишь один раз, невозможно сделать так, чтобы данная ссылка ссылалась на другой объект)

```
double  agent = .028
double  &bond = agent;  // bond ссылается на agent
// ...
bond /= 4.0;  // agent == .007
```

- Повышают эффективность программы при передаче больших объектов в функции, поскольку не требуют

*В C преобразование типов не требуется.

копирования объекта в стек*.

```
// String - это класс, определенный в <String.h>

char LastChar( const String &s )
{
  return s.length() ? s[ s.length() - 1 ] : '/0';
}
// ...
String word = "barbecue";
char c = LastChar( word );
```

- Предоставляют функциям механизм изменения значения передаваемых им аргументов.

```
inline void Swap( int &a, int &b )
{
  int t = a;
  a = b;
  b = t;
}
// ...
Swap( x, y );  // Не требуется передавать адреса
```

- Главным образом используются при определении функций для типа class.

- Не являются самостоятельным типом и существуют только после инициализации. Какие-либо операции могут выполняться не над ссылками, а над объектами, на которые они ссылаются.

*При использовании ссылок исключительно в целях повышения эффективности, а не для изменения значений аргументов, ссылки надо объявлять с использованием ключевого слова const. При этом компилятор будет следить за тем, чтобы значение ссылки не изменялось.

- Не могут ссылаться ни на другую ссылку, ни на битовое поле (см. "Структуры"). Кроме того, не может быть ни массивов ссылок, ни указателей на ссылку*.

Константы...

- Это объекты, значение которых после инициализации не может быть изменено (это должно быть сделано во время инициализации)**.

```
const long double pi = 3.14159265358979323846;
const int prime[ ] = { 1, 2, 3, 4, 5, 7, 11 };
```

- Для глобальных констант по умолчанию область видимости ограничивается файлом, в котором они определены. Для доступа к таким константам из других файлов при объявлении необходимо использовать префикс extern***.

```
extern const int prime[ ] = { 1, 2, 3, 5, 7, 11 };
```

- При использовании указателей распространяются как на сам указатель, так и на объект, адресуемый этим указателем. И то, и другое, в этом случае является константой.

```
char a, b;

const char *pcc = &a;  // Указатель на const char
```

*Однако может быть объявлена ссылка на указатель. Например, int *&rpi - это ссылка на указатель на int.

**Наличие констант позволяет не пользоваться директивой препроцессора #define, как это делается в C.

***В ANSI C глобальные константы по умолчанию имеют внешнюю область видимости.

```
pcc = &b;  // Правильно
*pcc = 'Z'; // Неправильно

char *const cpc = &a; // Константный указатель на char
cpc = &b;             // Неправильно
*cpc = 'Z';           // Правильно

const char *const cpcc = &a; // Обе константы
cpcc = &b;                   // Неправильно
*cpcc = 'Z';                 // Неправильно
```

Регистровые переменные...

- Могут повышать скорость программы при доступе к автоматическим переменным*.
- Обозначаются при помощи префикса register, используемого при объявлении.

```
register int i, *pint;
```

- Могут хранить любой тип данных, который можно разместить в машинном регистре. Для остальных типов префикс register игнорируется.
- Имеют ограниченные ресурсы. Не гарантируется, что переменная объявленная, как регистровая, будет помещена в регистр.
- К ним может применяться операция снятия ссылки & (не так, как в C), но при этом спецификация register будет игнорироваться.

---

*Явное описание регистровых переменных становится, если еще не стало, ненужным. Многие компиляторы достаточно умны для того, чтобы самостоятельно определять, какие переменные необходимо сделать регистровыми. Поэтому спецификация register - это не более чем "совет" компилятору, который он, компилятор, волен проигнорировать.

Переменные volatile...

- Обычно используются для ограничения выполняемой компилятором отимизации использования переменных, которые могут быть изменены извне программы*.

```
// serialPort - это расположенная где-то в памяти
// область ввода/вывода
voilatile char * const serialPort = ( char * ) 0xa400;

char c = *serialPort;  // Прочитать последовательный
                       // порт
c = *serialPort;       // Прочитать снова
```

Без объявления voilatile оптимизирующий компилятор может удалить второе присвоение значения переменной c, поскольку с его точки зрения этой переменной дважды присваивается одно и то же значение.

Структуры...

- Это набор связанной информации, возможно, различных типов, объединенный в один объект.

- Могут иметь имя, которое, в этом случае, будет именем типа**.

```
struct Call
{
  short area, exchange, line;
  enum{ direct, oper_assist } type;
  enum{ day, evening, weekend } rate;
```

*Использование описателя voilatile зависит от реализации компилятора. В некоторых компиляторах эта возможность вообще не реализована.

**Возможность опустить имя была предусмотрена для совместимости с C.

```
};
// ...
Call collect; // Объявить переменную типа Call*
```

- В случае, когда имеют статический класс памяти, могут быть инициализированы значениями, которые заключены в фигурные скобки и расположены в порядке объявления полей структуры. Если какие-то значения в конце списка останутся не инициализированы, то они инициализируются значением ноль (или эквивалентным).

```
Call work = { 708, 555, 8887 };**
// work.type = direct, work.rate = day
```

Статические векторы структур также могут быть инициализированы. Если число значений равно числу полей структуры, то внутренние скобки могут быть опущены.

```
Call record[ ] =
{                       // Для первых двух элементов
  { 516, 555, 8858 },  // type = direct
  { 508, 555, 0516 },  // rate = day
  // Внутренние скобки могут быть опущены, если все
  // элементы инициализированы
  212, 555, 5444, direct, weekend,
  908, 555, 6012, oper_assist, day
};
```

- Могут содержать *битовые поля*. Битовое поле - это раз-

*В C для определения переменной структурного типа было необходимо ключевое слово struct (если не используется typedef). В C++ это не обязательно, но допустимо.

**Пример объявления глобальной структуры.

дробленный целый тип*. Битовые поля позволяют экономить пространство, отводимое под данные**, а также осуществлять преобразование данных в другие форматы***.

Битовые поля без имени просто резервируют указанное число битов. Битовое поле без имени с нулевым размером указывает, что следующее битовое поле начинается на границе слова.

```
struct Features
{
  unsigned touchTone: 1;
  unsigned callWait: 1;
  unsigned callFwd: 1;
  unsigned: 2;                          // Не используется
  unsigned speedDial: 1;
  unsigned dstnctRng: 1;
};
```

Объединения...

- Это *нечто* различных типов, объединенное в одном объекте.
- Могут иметь имя, которое становится типом.
- Имеют размер, достаточный для размещения наибольшего из полей.

---

*Является ли в этом контексте тип int знаковым или беззнаковым, зависит от реализации компилятора. Для уверенности пожно явно объявить signed либо unsigned. ANSI C позволяет использовать только битовые поля типа int. C++ позволяет использовать любой целый тип.

**Однако это увеличивает размер кода и снижает производительность.

***Однако порядок размещения битовых полей в машинном слове зависит от реализации компилятора.

```
union Labor
{
  int UAW;
 char teamsters;
  char *CWA;
};
// ...
Labor day;  // Объявляет переменную Labor*
```

- Могут быть инициализированы (если имеют статический класс памяти) значениями, заключенными в фигурные скобки. Может быть инициализировано только первое поле.

```
Labor day = { 5 };**  // Правильно 5 - это int
Labor intensive = { 'a' };**  // Неправильно
```

Статические массивы объединений также могут быть инициализированы.

```
Labor record[ ] = { 3, 4, 5 };
```

- Могут быть анонимными, то есть не иметь имени***. К полям таких объединений можно обращаться как если бы они были самостоятельными переменными****.

---

*В C для определения переменной объединения было необходимо ключевое слово union (если не используется typedef). В C++ это не обязательно, но допустимо.

**Пример глобального объявления.

***Анонимные объединения поддерживаются только в C++.

****Глобальные анонимные объединения должны быть объявлены как static.

```
union
{
  long jekyll[100];
  char  *hyde[100];
};

hyde[0] = "Dr. Jekyll";  // Воздействует на jekyll[0]
```

Переименование типов...

- Вводит идентификатор, который является синонимом для существующего типа и предоставляет программисту более удобное название или сокращение данного типа.

- Осуществляется при помощи ключевого слова typedef, которое помещается перед "объявлением переменной". Однако вместо объявления новой переменной мы получаем синоним для объявляемого типа.

```
typedef float GIGAWATT;
typedef const char CHAR;
// ...
GIGAWATT future = 1.21;

int MyFunction( char );

// Указатель на функцию, принимающую char и
// возвращающую int
typedef int ( * PFCRI ) ( char );
//...
PFCRI myFunctionPtr = &MyFunction;
```

- Может быть произведено внутри класса (см. "Классы"). Новое имя типа имеет область видимости, ограниченную

данным классом.

```
class C
{
  // ...
public:
  // Указатель на функцию член класса C,
  // которая получает и возвращает int
  typedef int ( C::*PMFCIRI ) ( int );
  // ...
};
// ...
PMFCIRI p;    // Ошибка: PMFCIRI вне области
              // видимости
C::PMFCIRI q; // Правильно
```

Функции...

- Это наборы из нуля или более операторов, объединенных в исполняемый модуль, который выполняет действия, определенные программистом.

- Вызываются другими функциями и, как правило, возвращают им значение (если только возвращаемое значение не имеет тип void).

- Имеют несколько разновидностей: обычные (описываются здесь), члены (см. "Функции члены" в разделе "Классы"), дружественные (см. "Дружественные функции" в разделе "Классы") и шаблоны (см. "Шаблоны").

Объявление...

- Определяет имя функции*, тип возвращаемого значения

*В C++ все функции должны быть объявлены перед их использованием.

и типы аргументов*.

```
// Два аргумента типа double и возвращаемое
// значение типа double
double Hypotenuse( double a, double b );
```

- Имена аргументов могут быть опущены**

```
// То же, что и прежде, но с точки зрения компилятора
double Hypotenuse( double, double );
```

- Может указывать значения аргументов, используемые по умолчанию.

```
// Необязательный аргумент типа char, по умолчанию
// принимает значение '-'
void PrintLine( int length, char = '-' );
```

- Может быть *перегружено*, то есть одно и то же имя могут иметь несколько функций с различными типами или количествами параметров для которых не заданы значения по умолчанию***.

---

*В ANSI C запись f() означает, что функции передается ноль или более аргументов неопределенного типа. В C++ это означает, что аргументы функции не передаются. И в ANSI C, и в C++ запись f(void) означает, что аргументы функции не передаются.

**Однако их присутствие желательно, чтобы документировать функцию для пользователя.

***Для любого типа T записи T, T&, const T и volatile T для компилятора неразличимы и, следовательно, не могут быть использованы для различения перегружаемых функций. Не может быть использован для различения перегружаемых функций и тип возвращаемого значения. Однако записи const T& и const T* отличаются от T& и T* соответственно.

```
// Функции имеют два аргумента типа int или два
// аргумента типа double и возвращают
// соответственно int или double
double Max( double, double );
int Max( int, int );  // Правильно перегружена

// Один аргумент типа int, а другой имеет значение
// по умолчанию, но это неправильно
int Fluff( int, char = '*' );
int Fluff( int, int = 0 );  // Ошибка
```

- Может включать аргумент многоточие (...), указывающий компилятору, что вместо него могут быть подставлены ноль и более аргументов неопределенного типа*.

```
void printf( const char *format ... );
```

Определение...

- Описывает код, выполняемый при вызове функции.

- Может быть объявлено inline для оптимизации очень маленьких функций**. Это объявление не гарантирует, что функция, объявленная inline, на самом деле является таковой***.

- Может иметь свои аргументы, объявленные register.

- Для функций, которые имеют тип возвращаемого значения, отличный от void, должно иметь по крайней мере

---

*См. реализацию функций с произвольным числом аргументов неопределенного типа в пункте "stdarg.h" раздела "Другие библиотеки".

**В противоположность обычным функциям, функции inline помещаются в файлы заголовков. Функции inline являются альтернативой использованию директивы препроцессора #define.

***Поскольку эта возможность реализована далеко не везде.

один оператор return.

```
inline double Hypotenuse( double a, double b )
{
  return sqrt( a * a + b * b );
}

void PrintLine( register char ch )
{
  for( register int i = 0; i < 80; ++i )
  cout << ch;
  cout << endl;
}

inline double Max( double a, double b )
{
  return a > b ? a : b;
}

inline int Max( int a, int b )
{
  return a > b ? a : b;
}
```

- Могут иметь безымянные аргументы. Эту возможность можно использовать, когда необходимо оставить место в списке аргументов для будущих реализаций.

```
void Peek( unsigned int * p, int )
{
  cout << '*' << p << " = " << *p << '/n';
}
```

Вызов...

- Выполняет код, связанный с функцией.
- Выполняется двумя способами: прямо - по имени, или косвенно - через указатель.

```
c = Hypotenuse( a, b );          // Прямо
void ( *pfii ) ( int, int ) = &Max;
int x, y;
// ...
int m = ( *pfii )( x, y );       // Через указатель*
```

- Все аргументы передает по значению, то есть они вычисляются и копируются в стек. Имеют место два исключения:

  Если аргумент является вектором, то в функцию передается только адрес первого элемента этого вектора**.

  Если аргумент объявлен как ссылка, то его значение передается по ссылке, то есть вычисляется его адрес, который затем помещается в стек.

Функция main()...

- Необходима всем программам на C++.
- Это функция, с которой начинается выполнение программы.

---

*В этом примере компилятор знает, что указатель pfii указывает на int версию функции Max, поскольку pfii объявлен как указатель на функцию, которая получает два аргумента типа int и возвращает значение типа int.

**Фактически это не исключение, поскольку выражение, состоящее из имени вектора, вычисляет адрес первого элемента этого вектора (см. "Векторы" в разделе "Объявления").

- Имеет одну из двух форм аргументов:

```
int main() { /* .. */ }
int main( int argc, char *argv[]) { /* ... */ }
```

Последняя используется для ввода аргументов из окружения, в котором выполняется программа. Параметр argc - число аргументов. Параметр argv[ ] - вектор указателей на char, каждый из которых указывает на аргумент. argv[0] - это имя, которое использовалось для запуска программы, argv[1] ... argv[argc - 1] - это собственно аргументы.

```
$ magic hat wand*

argc = 3
argv[0] = "magic"
argv[1] = "hat"
argv[2] = "wand"
```

- Обычно возвращает в программу окружения код завершения программы при помощи оператора return (см. "exit()" в пункте "stdlib.h" раздела "Другие библиотеки").

- Не может быть перегружена, не может иметь аргументов со значениями по умолчанию, не может быть вызвана пользователем, не может получить свой адрес, не может быть объявлена inline или static**.

*В этом примере использован некоторый интерфейс с командной строкой, в котором команда вводится пользователем с клавиатуры. Символ $ - это приглашение системы.

**В C main() может быть вызван пользователем и может получить свой адрес.

Связь с C...

- Позволяет скомпилированные ранее функции на C, находящиеся в других файлах, вызывать из программ на C++*.

- Осуществляется путем объявления C-функций с использованием записи extern "C" в одной из представленных ниже форм**:

```
// Объявить одну функцию C
extern "C" int system( const char * );

extern "C"
{           // Объявить несколько функций C
  long   atol( const char * );
  int    atoi( const char * );
  double atof( const char * );
};
```

Обычно в стандартных файлах заголовков C эти объявления extern "C" вы должны сделать самостоятельно, чтобы использовать объявленные там функции в вашей программе на C++.

---

*Компилятор C++ кодирует имена функций, включая в них типы их аргументов для того, чтобы правильно осуществлять их вызов. Обычно это незаметно, однако, при использовании C-функций в программах на C++ компилятору нужно указать, что имена C-функций кодироваться не должны. В противном случае при связывании с объектными файлами, содержащими C-функции, произойдет ошибка. (Прим. переводчика: при работе с компиляторами Turbo C и Turbo/Borland C++ причина несоответствия состоит в том, что перед именами C-функций компилятор добавляет символ подчеркивания (_), а перед именами функций, написанных на C++ - нет.)

**Первые версии C++ не поддерживают этот механизм.

Классы...

- Предоставляют средства для создания совершенно нового типа, который может быть полностью интегрирован в язык.

- Это наборы связанной информации, возможно различных типов (данные-члены класса, поля класса), объединенной в один объект вместе с методами для обработки хранимых данных (функции-члены класса, методы класса).

Объявление...

- Вводит новый тип.

- Устанавливает ограничение доступа к членам класса. Для этого используются следующие ключевые слова: private, protected и public. До спецификации присваивается private.

  Private означает, что члены класса доступны только функциям-членам класса и дружественным функциям класса (см. "Дружественные функции").

  Protected означает, что члены класса доступны лишь членам класса и дружественным функциям производных классов* (см. "Производные классы").

  Public означает, что члены класса доступны любым функциям.

- Похоже на объявление структуры (см. "Структуры" в разделе "Объявления").

*Первые версии C++ не имеют членов с доступом protected.

```
class Phone
{
  int area, exchange, line;  // private
protected:
  enum HookState { onHook, offHook } handset;
public:
  Phone( int a, int e, int l )
  {
    area = a, exchnge = e, line = l;
    handset = onHook;
  }
  int   OffHook() const { return hanset == offHook; }
  void GiveDailTone();
  long AcceptDigits();
  void Ring() const;
};

// Объявить переменную класса
Phone home( 516, 555, 8858 );
```

- Может содержать объявление другого класса. Область видимости вложенных классов ограничивается границами класса, в котором они объявлены*.

*В ранних версиях C++ вложенные классы не являются локальными для классов, в которых они определены.

```
class OuterShell
{
  int oa;
public:
  int ob;
  class InnerSelf
  {
    int ia;
  public:
    int ib;
    void Vague( OuterShell& o, int n )
    {
      oa = n;  // Ошибка: какое oa?
      o.oa = n;  // Ошибка: oa - private
      o.ob = n;  // Правильно: ob - public
    }
    void Later();  // Определена ниже
  };
  void Vague( InnerSelf& i, int n )
  {
    ia = n;  // Ошибка: какое ia?
    i.ia = n;  // Ошибка: ia - private
    i.ib = n;  // Правильно: ib - public
  }
};

InnerSelf shallow;  // Ошибка: вне области видимости
OuterShell::InnerSelf deep;  // Правильно

// Как определить функцию-член класса InnerSelf
void OuterShell::InnerSelf::Later() { /* ... */ }
```

- Могут содержать локальные перечисления и переименования типов typedef (см. "Перечисления" и "Переименование типов" в разделе "Объявления").

- Могут быть локальными в функциях. Такие классы не могут использовать автоматические переменные в тех функциях, где они объявлены.

  Функции-члены должны быть определены в объявлении класса (inline). (см. "Функции-члены")

  Локальные классы не могут содержать статические поля (см. "Данные-члены").

```
void Contrived()
{
  int i;
  static s;

  classLocal
  {
    // ...
  public:
    int f() { return i; }  // Ошибка: i - автоматическая
                           // переменная
    int g() { return s; }  // Правильно: s - статическая
                           // переменная
  };
  // ...
}

Local vacuous;  // Ошибка: Local вне области
                // видимости
```

Данные-члены...

- Это набор взаимосвязанной информации, возможно различных типов, объединенной в один объект.

- Могут находиться в закрытой (private), защищенной

(protected) или открытой (public) части класса.


```
home.area = 708;*          // Ошибка: private
home.handset = offHook;  // Ошибка: protected
if( home.OffHook() )       // Правильно
 home.GiveDialTone();
```

- Могут иметь статический класс памяти (static). Поля, имеющие статический класс памяти, совместно используются всеми объектами класса.

  К ним возможен доступ через имя класса (с использованием операции разрешеня доступа), а не через конктетный объект класса.

  Статические поля должны быть определены в области видимости файла. Они могут быть инициализированы; если нет, то инициализируются значением ноль**.


```
// Файл Shape.h
class Shape
{
  static int  numShapes;  // Объявление
  // ...
};

// Файл Shape.c
int Shape::numShapes = 0;  // Определение
```

- Могут быть объявлены как const. Константные данные-члены класса должны быть инициализированы в каждом

*Предполагается, что в примере приведен код функции, которая не является функцией-членом данного класса или дружественной функцией.

**Ранние версии C++ не поддерживают инициализацию и определение статических полей.

определении конструктора* (см. "Конструкторы"). Имена полей и их начальные значения, заключенные в скобки, отделяются от списка аргументов конструктора двоеточием.

```
class  Phone
{
  const int area, exchange, line;
  // ...
public:
  Phone( int a, int e, int  l ) :
            area( a ), exchange( e ), line( l )
  {
   // ...
   }
  // ...
};
```

- Могут быть переменными другого класса. В этом случае требуется явная инициализиция, если только поля не имеют конструктора по умолчанию (см. "Конструкторы"). Такие поля инициализируются тем же способом, что и константные поля.

```
class Phone
{
  // ...
public:
  Phone( int, int, int );  // Нужны аргументы.
  // ...
};
```

*Классы с константными полями имеют по крайней мере один конструктор для их инициализиции. Тело конструктора может содержать только фигурные скобки { }.

```
class Car
{
  // ...
  Phone phone;
public:
  Car( /* ... */, int a, int e, int l ):
      phone( a, e, l ) { /* ... */ }
  // ...
};
```

Функции-члены...

- Это функции, которые манипулируют данными-членами класса.
- Имеют доступ ко всем полям своего класса.
- Могут быть в закрытой, защищенной или открытой части класса.
- Могут быть определены внутри или вне объявления класса в собственном базисе*. Функции-члены класса, определенные вне класса, могут быть сделаны inline**. Однако таким функциям не могут быть переданы аргументы по умолчанию.
- Могут обращаться к полям или функциям-членам, объявленным после них в объявлении данного класса.

---

*Небольшие по объему функции-члены обычно помещаются в объявление класса. Большие функции-члены чаще всего помещаются в файлы с расширением .c. Окончательное решение принимает пользователь, однако реализации языка могут накладывать ограничения на то, какие типы операторов могут использоваться в функциях-членах типа inline. В этом случае пользователь должен поместить определение такой функции в файл с расширением .c.

**Объявление функции внутри класса может предваряться ключевым словом iniine. Это делается лишь для ...шения читабельности программы.

```
// Файл IntStack.h
class IntStack
{
  int * v, size, top;
public:
  // ...
  int   Top() const;  // Отложен
  void Push( int );  // Тоже
};

inline  int IntStack::Top() const { return v[top]; }

// Файл IntStack.c
void IntStack::Push( int element )
{
  if( top+1 < size ) v[++top] = element;
}
```

- Имеют неявно объявленную переменную this, которая представляет собой указатель на объект класса, членом которого является данная функция. В большинстве случаев в использовании этого указателя не возникает необходимости. Однако он бывает полезен, например, при реализиции связанных списков.

```
class Node
{
  // ...
  Node   * next;
public:
  // ...
  void InsertAfter( Node * p )
  {
    if( this && p )  // Проверка на ноль
    {
```

```
      next = p->next;  // next - это this->next
      p->next = this;  // Явное использование this
    }
  }
};
```

Кроме того, этот указатель позволяет осуществить вызов цепочки функций-членов данного класса. Это можно сделать, возвращая ссылку на объект класса *.

```
class Phone
{
  // ...
public:
  Phone& GiveDialTone();
  // ...
};

Phone& Phone::GiveDialTone()
{
  // ...
  return *this;  // *this - это Phone&
}

Phone home( 516, 555, 8858 );

long digits = home.GiveDialTone().AcceptDigits();
// Сначала выполняется...
// home.GiveDialTone(); а затем...
// long digits = home.AcceptDigits();
```

- Могут быть объявлены static. Такие функции могут толь-

* Также можно использовать указатель this в конструкторах (см. "Конструкторы") и при перегрузке операции присваивания (см. "Операция присваивания" в разделе "Перегрузка операторов").

ко непосредственно обращаться и изменять статические поля класса*. Статические функции-члены класса не могут быть объявлены const или virtual**.

К таким функциям можно обращаться через имя класса (используя операцию разрешения доступа), а не через имя конкретного экземпляра объекта класса.

```
class Shape
{
  static int numShapes;
protected:
  Point center;
public:
  Shape( Point c ) : center( c ) { ++numShapes; }
  ~Shape() { --numShapes; }
  // ...
  static int NumShapes() { return numShapes; }
};
// ...
for( int n = Shape::NumShapes(); n > 0; --n )
// ...
```

При определении вне объявления класса static не переопределяется.

```
class Shape
{
  static int numShapes;  // Определено ниже
};
```

*Ранние версии C++ не поддерживают статических функций-членов.

**Из этого следует, что статические функции-члены не имеют указателя this, поскольку они не связываются с конкретным экземпляром объекта данного класса.

```
// ...

inline int Shape::NumShapes()  // Здесь static нет
{
  return numShapes;
}
```

- Могут быть объявлены как const, что не позволяет им изменять значение и возвращать неконстантную ссылку или указатель на любое поле класса*. Такие функции не могут быть статическими.

  Для константных объектов могут быть вызваны только константные функции-члены (они могут быть вызваны также и для неконстантных объектов).

```
class IntStack
{
  int  * v, size, top;
public:
  // ...
  int Pop();
  int Top() const;  // Не может изменять что-либо
};

void f( const IntStack& s )  // Константный объект
{
  int a = s.Top()  // Правильно: Top() - константный член
  int b = s.Pop();  // Ошибка: s не может быть
                    // модифицирован
}
```

- Могут быть объявлены volatile. Для объектов volatile могут быть вызваны только volatile функции-члены. Функ-

*Ранние версии C++ не поддерживают константных функций-членов.

ции-члены могут быть одновременно const и volatile.

- Могут быть виртуальными (см. "Виртуальные функции-члены" в разделе "Производные классы"). Виртуальные функции-члены не могут быть объявлены как статические.

Указатели на членов...

- Указывают на нестатические поля или методы (включая виртуальные) любого объекта класса* (Указатели на статические поля - это обычные указатели).

```
struct Point  // см. "Классы-структуры и
              // классы-объединения"
{
  int x , y;
  void Set( int x0, int y0 ) { x = x0, y = y0; }
};

Pointa a, *pa = &a;

// pint - это указатель на любой int член класса Point
int Point::*pint = &Point::x;  // В этом случае x

// pmf2i - это указатель на любую функцию-член
// класса Point, принимающую два аргумента типа
// int и возвращающую void
void ( S::*pmf2i ) ( int , int ) = &S::Set;

++( a.*pint );  // Инкремент x
++( pa->*pint );  // Снова инкремент x
( a.*pmf2i ) ( 1, 1 );  // Вызов Set()
```

*Указатель на поле int - это не указатель на определенный int, а смещение в объекте данного класса для указанного поля. Ранние версии C++ не поддерживают указатели на членов.

```
( pa->*pmf2i )( 2, 2 ); // Вызов Set() через pa
```

- Неявно преобразуются в указатели на членов производных классов, если это необходимо (см. "Указатели" в разделе "Производные классы").

- Не преобразуются неявно в указатели на void, как обычные указатели.

Конструкторы...

- Инициализируют объект класса с необязательным списком аргументов. Аргументы, если они используются, могут иметь значения по умолчанию.

- Автоматически вызываются компилятором при объявлении переменной данного класса, при необходимости преобразований в выражениях, аргументах функций или возвращаемых значениях, а также при создании динамической переменной при помощи операции new.

- Все невиртуальные конструкторы базовых классов, если таковые имеются, вызываются до них (см. "Конструкторы базовых классов" и "Виртуальные базовые классы" в разделе "Производные классы").

- Имеют то же имя, что и класс, для которого они определены.

- Может быть и в закрытой, и в защищенной, и в открытой части класса*.

*Если конструктор находится в закрытой части класса, то объект этого класса с использованием такого конструктора могут создавать только дружественные функции. Если конструктор находится в защищенной части класса, то объект этого класса с использованием такого конструктора могут создавать только дружественные функции или функции-члены производных классов.

- Может быть определен внутри или вне объявления класса (как и функция-член).

```
class IntStack
{
  int * v, size, top;
public:
  IntStack( int ezis ); // Объявление
  // ...
};

IntStack::IntStack( int ezis )
{
  v = new int[ size = ezis ];
  top = -1;
}

// Объявлене IntStack. Вызывается конструктор
IntStackhuge( 1000 );
```

- Не может быть объявлен ни const, ни volatile, ни static, ни virtual. Не имеет возвращаемого значения.

- Могут быть вызваны тремя эквивалентными способами (пользователю предоставляется свобода выбора).

```
IntStack good( 100 );
IntStack bad = IntStack( 100 ); // Явный
IntStack ugly = 100; // Сокращенный
```

- Выполняют определенные пользователем преобразования типов, если имеют один аргумент (либо если имеют только один аргумент, либо если остальные аргументы имеют значения по умолчанию).

Такие преобразования являются дополнением к арифметическим преобразованиям в выражениях при инициализации, передаче аргументов функциям и возвращении функциями результата (см. "Арифметические преобразования" в разделе "Операции").

Как правило, выполняется одно преобразование либо при помощи конструктора, либо при помощи операции преобразования (см. "Операции преобразования").

```
class String { /* ... */ String( const char * ); };
class BitString{ /* ... */ BitString( String ); };

void f( const String& s ) { /* ... */ }
void g( const BitString& b ) { /* ... */ }
// ...

  f( "Hello" );  // Преобразование String( "Hello" )
  g( "10110" );  // Ошибка: не происходит
// преобразование BitString( String( "Hello" ) )
```

- Используются для инициализации константных полей и полей, которые являются переменными класса*. Константное поле должно быть инициализировано в каждом определении конструктора**. Имя поля, заключенное в скобки, отделяется от списка аргументов конструктора двоеточием (см. также "Данные-члены").

---

*Таким образом могут быть инициализированы и неконстантные поля. Окончательный выбор делает пользователь, однако такой способ инициализации более эффективен.

**Классы с константными полями должны иметь по крайней мере один конструктор. Тело такого конструктора может содержать только фигурные скобки { }.

```
class IntStack
{
  const int size;
  // ...
};

IntStack::IntStack( int ezis ) : size( ezis )
{
  v = new int[ size ];
  top = -1;
}
```

- Конструкторы, которые были рассмотрены, - это *конструкторы по умолчанию*, если они не имеют аргументов (либо аргументов нет, либо все аргументы имеют значения по умолчанию)*.

  Конструктор по умолчанию требуется, когда создается массив объектов данного класса с использованием оператора new.

```
// p - это указатель на массив из пяти объектов
// класса IntStack
IntStack * p = new IntStack[ 5 ]; // Неправильно

class IntStack  // Модифицируем класс IntStack
{
  // ...
  IntStack( int = 20 );  // Конструктор по умолчанию
  // ...
};
```

*Если конструктор класса не определен, то компилятор автоматически генерирует конструктор по умолчанию, который вызывает конструкторы всех классов-полей и базовых классов, если таковые имеются.

```
// Каждый элемент массива имеет размер 20
IntStack * p = new IntStack[ 5 ];  // Теперь правильно
```

- Рассмотренные конструкторы являются *конструкторами копирования*, если они имеют один аргумент, являющийся ссылкой или константной ссылкой на данный класс (либо один аргумент, либо остальные аргументы имеют значение по умолчанию)*.

  Конструктор копирования служит для создания объекта из другого объекта того же класса**.

  Конструктор копирования должен быть определен, если класс содержит поля, являющиеся указателями на динамически распределяемую память***.
- Могут быть перегружены.

```
class IntStack
{
  // ...
public:
  IntStack( int = 10 );
  IntStack( const IntStack& );
  // ...
```

---

*То есть он записывается как X (X&) или X (const X&), причем последняя запись предпочтительнее. Если для класса не определен конструктор копирования, то компилятор автоматически создает конструктор копирования по умолчанию, который копирует все поля копируемого объекта.

**Кроме того, этот конструктор вызывается при передаче аргументов функциям и возврате значений по значению (а не через указатель).

***При отсутствии конструктора копирования, конструктор копирования по умолчанию сделает так, что указатель b объекта v и указатель b объекта a будут иметь одинаковые значения. По той же причине необходимо перегружать операцию присваивания, если конструктор копирования определен пользователем (см. "Операция присваивания" в разделе "Перегрузка операторов"). Инициализация объекта из другого объекта того же класса может быть запрещена, если конструктор копирования просто *объявить* в закрытой части класса.

```
};

IntStack::IntStack( const IntStack& from )
{
  v = new int[ size = from.size ];
  for( register int i = 0; i < size; ++i )
      v[i] = from.v[i];
  top = from.top;
}
// ...
IntStack a( 100 );
IntStack b( a );  // Объявить b, которое равно a
IntStack c = a;  // То же в альтернативной форме
```

- Могут инициализировать массивы объектов класса таким же образом, как и массивы встроенных типов*.

  Максимальное число элементов может быть опущено, если оно равно числу инициализирующих значений. Если максимальное число элементов больше числа значений, то оставшиеся значения инициализируются при помощи конструктора по умолчанию**. Если заданы все значения для данного размера массива, то фигурные скобки при указании списка значений могут быть опущены.

```
Phone office[] =
{                  // Компилятор вычислит размер за нас
  708, 555, 8887,
  708, 555, 7255
};
Phone office[3] =
{                        // Требуется конструктор
```

*Ранние версии C++ не допускают явной инициализации элементов.

**Если это не конструктор по умолчанию, то все значения должны быть указаны.

```
  708, 555, 6012,  // по умолчанию, которого
  708, 555, 3707   // Phone не имеет
};
```

Деструкторы...

- Выполняют все необходимые действия перед разрушением объекта.

- Автоматически вызываются компилятором:

  - при выходе из области видимости;
  - при создании временных объектов для преобразования в выражениях или для передачи функциям аргументов;
  - когда возвращенные функцией значения более не нужны;
  - при выполнении операции delete для объектов, размещенных в динамической памяти.

- Имеют то же имя, что и класс, к которому впереди добавлен значек "тильда" (~). Не имеют возвращаемого значения и не получают аргументов, следовательно не могут быть перегружены.

- Не могут быть виртуальными (см. "Виртуальные функции-члены" в разделе "Производные классы").

```
class IntStack
{
  int  * v, size, top;
public:
  IntStack( int esiz = 20 );
  ~IntStack() { delete[ ] v };  // Деструктор
  // ...
};
```

```
// ...
int main()
{
  IntStacks( 100 );
  IntStack * ps = new IntStack( 50 );
  // ...
  delete ps;  // Вызывается деструктор для ps
   // Неявно вызывается  деструктор для s
}
```

- Может быть явно вызван с использованием операции выбора члена* (в этом редко бывает необходимость).

```
IntStack s( 100 ), * ps = new IntStack( 50 );
//...
s.~IntStack();     // Явное разрушение s
this->~IntStack(); // Явное разрушение *this
```

Операции преобразования...

- Предоставляют механизм преобразования типов class во встроенные типы или предопределенные типы class.

- Имеют форму...

operator *type*() { /* ... */ }

...то есть не имеют ни явных аргументов, ни возвращаемого значения, следовательно не могут быть перегружены**.

---

*Ранние версии требуют, чтобы доступ к деструктору осуществлялся явно через операцию разрешения доступа. Первые версии вообще не поддерживают этот механизм.

**Может быть несколько операторов преобразования для различных типов.

- Используются как дополение к преобразованиям в арифметических выражениях, инициализациях, аргументах функций и возвращаемых значениях (см. "Арифметические преобразования" в разделе "Операции").

```
class String
{
  char  * s;
public:
  // ...
  operator const char * () const { return s; }
};

void f( const char * s ) { /* ... */ }

String s;
// ...
  f( s );  // Преобразование: s.operator const char * ()
```

В большинстве случаев преобразование выполняется либо при помощи оператора преобразования, либо при помощи конструктора (см. "Конструкторы").

```
class BitString { /* ... */ operator String(); };
BitString b;
// ...
  f( b );  // Ошибка: преобразование не выполняется
// b.operator String().operator const char * ()
```

- Наследуются и могут быть виртуальными (см. "Виртуальные функции-члены" в разделе "Производные классы").

Классы-структуры и классы-объединения...

- Полностью похожи на обычные классы, за исключением того, что в объявлении используют ключевые слова struct или union.

- Это классы, в которых по умолчанию все члены и базовые классы являются public, в отличие от обычных классов, в которых по умолчанию используется private*.

```
// Это два одинаковых класса

class Ture : public X
{
  int a;
protected:
  int f();
public:
  Ture();
  int g();
};

class Imitator : X
{
  Imitator();
  int g();
protected:
  int f();
private:
  int a;
};
```

*Выбор той или иной формы объявления класса должен сделать программист. Если все члены класса находятся в открытой части, более предпочтительным является описание его в качестве класса-структуры.

- Классы-объединения ведут себя как обычные объединения, то есть это набор различных типов, из которых в каждый момент времени может храниться только один.

  Классы-объединения не могут иметь базовые классы или использоваться в качестве базовых классов, следовательно не могут иметь ни виртуальных функций-членов, ни статических полей. Анонимные классы-объединения не могут иметь ни полей, ни функций-членов, которые являются закрытыми или защищенными.

Дружественные функции...

- Это функции, которые имеют доступ к закрытой части класса, членами которого они не являются, и защищенной части класса, от которого они не порождены.

- Могут быть дружественными более чем одному классу.

- Объявляются внутри объявления класса, для которого являются дружественными*. Типы аргументов и тип возвращаемого значения должны полностью совпадать с объявленными.

```
classPerson
{
  int secret;  // закрытая часть класса
public:
  // ...
  friend void Spouse( Person& );  // Не член
};
// ...
void Spouse( Person& p )
{
```

*Безразлично, объявлена функция в закрытой, защищенной или открытой части класса.

```
  ++p.secret;  // Имеет доступ к закрытой части класса
};
```

- Могут быть членами другого класса.

```
class Person
{
  // ...
  friend void Family::Sibling();
  // Теперь Family::Sibling() - дружественная функция
};
```

- Могут быть объявлены для всего класса.

```
class Person
{
  // ...
  friend class Family;
  // Все функции-члены класса Family являются
  // дружественными
};
```

- Не транзитивны, то есть друзья друзей класса не являются друзьями данного класса.

```
class Family
{
  friend class DistantRelative;
  // ...
};

class DistantRelative : public Family
{
  void Pry( Person& p )
```

```
  {
    ++p.secret;  // Ошибка
    // DistantRelative не является дружественным
    // классу Person, хотя он и порожден от
     // дружественного ему класса
  }
};
```

- Не наследуются производными классами (см. "Производные классы").

```
class DistantRelative : public Family
{
  void Pry( Person& p )
  {
    ++p.secret;  // Ошибка
    // DistantRelative не является дружественным
    // классу Person, хотя он и порожден от
     // дружественного ему класса
  }
};
```

Производные классы...

- Предоставляют средства для создания класса, который является вариацией класса, определенного ранее.
- Имеют доступ к открытой и защищенной части базового класса.

Объявление...

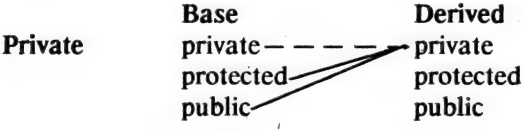
- Вводит новый тип.
- Устанавливает ограничение доступа private, protected или public для членов своего базового класса. По умолчанию устанавливается private.

  Private - открытые и защищенные члены базового класса в производном классе становятся закрытыми.

  Protected - открытые и защищенные члены базового класса в производном классе становятся защищенными*.

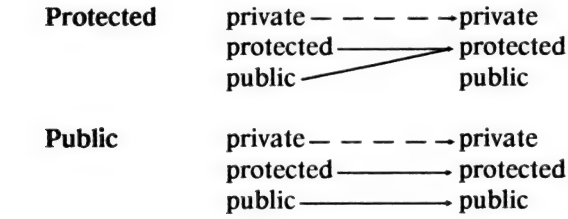
  Public - открытые и защищенные члены базового класса в производном классе становятся открытыми и защищенными соответственно.

  Графически эти правила представлены на рисунке**:



*Защищенные базовые классы поддерживаются, начиная с версии C++ USL 3.0.

**Пунктирная линия означает, что хотя закрытые поля базового класса и становятся частью производного класса, но к ним нельзя получить доступ через производный класс.



- Полностью совпадает с объявлением обычного класса, но кроме того, содержит отделенный двоеточием список своих базовых классов с их ограничением доступа.

```
class Phone
{
  // См. объявление класса Phone в разделе "Классы"
  //...
};

// Phone является базовым классом public для
// класса PayPhone
class PayPhone : public Phone
{
  int centsDeposited;
public:
  PayPhone( int area, int exchange, int line );
  void GiveDialTone();
  int  AcceptCoins();
};

// Объявление переменной производного класса
PayPhone booth( 708, 555, 5444 );
```

Функции-члены...

- В порожденных классах могут перекрывать функции-члены базового класса с тем же именем.

Число и типы аргументов, а также тип возвращаемого значения функции-члена производного класса могут отличаться от числа и типов аргументов и типа возвращаемого значения перекрываемой функции базового класса.

```
class PayPhone : public Phone
{
  // ...
public:
  // ...
  void GiveDialTone();  // Как обявлено ранее
};

Phone home( 516, 555, 8858 );
PayPhone booth( 708, 555, 5444 );

// ...

home.GiveDialTone();  // Phone::GiveDialTone();
booth.GiveDialTone(); // PayPhone::GiveDialTone();
```

- Могут вызывать функции-члены базового класса, используя операцию разрешения доступа.

```
void PayPhone::GiveDialTone()
{
  // Выполняет что-то особенное для PayPhone...
  Phone::GiveDialTone();  // Вызов оригинала*
}
```

*Если этого не сделать, то произойдет рекурсивный вызов.

Объявления доступа...

- Дают возможность сделать опять защищенными или открытыми защищенные и открытые члены закрытого базового класса в производном классе соответственно.

- Позволяют сделать снова открытыми открытые члены базового класса в защищенном производном классе*.

- Для перегруженных функций позволяют вернуть им первоначальное ограничение доступа, которое они имели в базовом классе, если все они имели одинаковое ограничение доступа.

```
class Base
{
  int   a;
  void g();
public:
  int b, c;
  void f(), f( int ), g( int );
};

class Derived : private Base
{  // ...
public:
  Base::a;// Ошибка: нельзя сделать 'a' public
  Base::b;// Вновь делает b public
  int c;
  Base::c;// Ошибка: ошибка с объявляется дважды
  Base::f;// Вновь делает все f() public
  Base::g;// Ошибка: функции g() имеют различное
          // ограничение доступа
};
```

*Ни этот, ни предыдущий пункты не выполняются, если в производном классе объявлены члены с теми же именами, что и в базовом классе.

Указатели и ссылки...

- Для базового класса указывают или ссылаются на объект производного класса (см. пример в параграфе "Виртуальные функции-члены").

Виртуальные функции-члены...

- Позволяют выбирать члены с одним и тем же именем через указатель функции в зависимости от *типа объекта, на который указывает указатель*, а *не* в зависимости от типа указателя.

- Позволяют использовать *объектно-ориентированную* парадигму программирования.

- Типы аргументов, их количество, а также тип возвращаемого значения должны быть такими же, как у одноименной функции в базовом классе.

- Не могут быть статическими.

```
class Phone
{
  // ...
public:
  virtual void GiveDialTone(); // Виртуальная функция
  // ...
};

class PayPhone : public Phone  // Без изменений
{
  int centsDeposited;
public:
  PayPhone( int area, int exchange, int line );
  void GiveDialTone();
```

```
  int AcceptCoins();
};

// ...

Phone home( 516, 555, 8858 );
payPhone booth( 708, 555, 5444 );
Phone * p;  // Указатель на любой вид Phone

p = &home;  // Здесь никаких сюрпризов нет...
p->GiveDialTone();  // Вызов Phone::GiveDialTone()

p = &booth;  // А здесь нечто таинственное...
p->GiveDialTone();  // Вызов PayPhone::GiveDialTone()
```

- Могут использоваться для создания *абстрактных базовых классов*[*]. Абстрактный базовый класс - это класс, который содержит по крайней мере одну *полностью виртуальную функцию-член*. Полностью виртуальная функция-член - это виртуальная функция-член, для которой объявлен интерфейс, а реализация находится в производном классе[**]. Для этого надо сделать объявление функции-члена равным нулю.

```
struct Point
{
  int x, y;
  Point( int x0=0, y0=0 ) { x = x0, y = y0 }
};
```

[*] Деструктор абстрактного класса всегда должен быть виртуальным. Ранние версии C++ не поддерживают абстрактные базовые классы.

[**] Невозможно объявить объект абстрактного класса. Может быть объявлен лишь производный объект, в котором определены все полностью виртуальные функции базового класса.

```
class Shape  // Абстрактный базовый класс
{
protected:
  Point center;
public:
  Shape( Point c ) : center( c ) { }
  virtual Shape& Draw() const = 0;  // Полностью
                                    // виртуальная
};

class Circle : public Shape
{
  int radius;
public:
  Circle( Point c, int r = 1 );
  Shape& Draw() const;  // Draw() будет определено
};
// ...
Shape amorphus;  // Ошибка: абстрактный класс
Circle c;
Shape  * p = &c;  // Правильно: указатель на любой
                  // Shape
```

Полностью виртуальная функция никогда не может быть явно или косвенно вызвана из конструктора.

- Может быть вызвана из конструктора прямо или косвенно через невиртуальную функцию-член класса. Однако будет вызвана версия функции, определенная в данном классе, либо версия функции, определенная в базовом классе, если таковой имеется, но не версия функции, определенная в производном классе.

```
class Phone
{
  // ...
```

```
public:
  Phone( int a, int e, int l )
  {
    // ...
    AcceptDigits(); // Эта функция вызывает...
  }
  virtual void GiveDialTone(); // ...эту функцию...
  long AcceptDigits()
  {
    // ...
    GiveDialTone(); // ...поскольку прямой вызов
                    // находится здесь, а...
  }
// ...
};

class PayPhone : public Phone
{
  // ...
public:
  // ...
  void GiveDialTone(); // ...эта функция вызвана быть
                       // не может
};
```

Конструкторы базовых классов...

Вызываются перед вызовом конструкторов производных классов.

Должны быть явно указаны в каждом определении конструктора производного класса, если у базового класса нет конструктора по умолчанию*.

*Это значит, что если даже производный класс не требует для себя конструктор, то он все равно должен иметь его для вызова конструкторов базовых классов, которые имеют аргументы. Тело такого конструктора производного класса может содержать только фигурные скобки.

Имя конструктора базового класса* и его аргументы, заключенные в скобки, отделяются от имени конструктора производного класса двоеточием**. Конструкторы множественных базовых классов отделяются друг от друга двоеточием (см. "Множественные базовые классы").

```
class Phone
{
  // ...
public:
  Phone( int, int, int );  // Имеет аргументы
  // ...
};

class PayPhone : public Phone
{
  // ...
public:
  PayPhone( int a, int e, int l ) : Phone( a, e, l )
  {
    // ...
  }
  // ...
};
```

- Могут через невиртуальные функции-члены вызывать прямо или косвенно не полностью виртуальные функции-члены. Однако будет вызываться версия виртуальной функции для данного класса или для его базового класса, если таковая существует, и никогда - для призводных классов (см. "Виртуальные функции-члены").

- Никогда не должны прямо или косвенно вызывать пол-

*Ранние версии C++ не позволяют указывать имя.

**В точности как константные поля (см. "Данные члены" в разделе "Классы").

ностью виртуальные функции-члены (см. "Виртуальные функции-члены").

Деструкторы базовых классов...

- Вызываются после деструкторов производных классов.
- Могут быть виртуальными. Деструктор абстрактного класса всегда должен быть виртуальным.
- Могут через невиртуальные функции-члены прямо или косвенно вызывать виртуальные функции-члены. Однако при этом вызывается версия функции для данного класса или для базового класса, если таковая имеется, и никогда - для производных классов.

Множественные базовые классы...

- Позволяют порождать класс от более чем одного класса*. Порядок представления базовых классов в списке не существенен**.
- К членам базовых классов, имеющим одинаковые имена, доступ должен осуществляться через имена базовых классов, которым они принадлежат, при помощи операции разрешения доступа.

```
class A { public: void f(); /* ... */ };
class B { public: void f(); /* ... */ };
class C : public A, public B { /* ... */ };
// ...
  C c;
  c.f(); // Ошибка: какое f(), из A или из B?
```

*Ранние версии C++ не поддерживают множественные базовые классы.

**Это существенно, если конструкторы или деструкторы базовых классов модифицируют глобальные данные.

```
c.A::f(); // Правильно
```

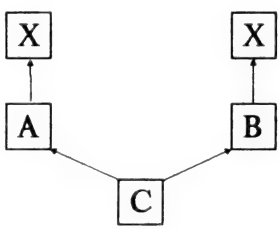
Наиболее удобно разрешать такую неоднозначность перекрытием в производном классе обеих функций.

```
class C : public A, public B
{
  // ...
public:
  void f() { A::f(); B::f(); } // Перекрываем обе функции
  //...
};
// ...
  C c;
  c.f(); // Теперь правильно
```

- Для классов, порожденных от производных классов с общей базой, по умолчанию существует два экземпляра объекта общей базы.

```
struct X { int i; /* ... */ X( int ); };

class A : public X { /* ... */ A( int ); };
class B : public X { /* ... */ B( int ); };
class C : public A, public B { /* ... */ C( int ); };
```

Графически это может быть представлено следующим образом:



К членам общего базового класса можно обратиться через имя одного из производных классов, используя оператор разрешения доступа.

```
C c( 0 );
++c.i;  // Ошибка: какое i, из A или из B?
++c.A::i;  // Правильно

C * cp = new C( 0 );
X * xp = cp;  // Ошибка: какое X?
X * xp = ( A * )cp;  // Правильно
```

Виртуальные базовые классы...

- Для классов, порожденных от производных классов с общим *виртуальным* базовым классом, существует только один экземпляр объекта общего базового класса.

- Объявляются включением ключевого слова virtual в спецификатор ограничения доступа базового класса*.

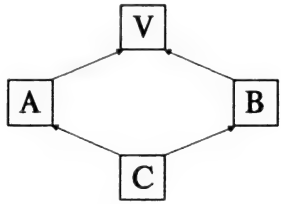
```
struct V { int i; /* ... */ V( int ); };

class A : virtual public V { /* ... */ A( int ); };
class B : virtual public V { /* ... */ B( int ); };
class C : public A, public B { /* ... */ C( int ); };
```

Графически это представлено на рисунке.

Для доступа к членам общего базового класса не требуется ничего указывать дополнительно, поскольку существует лишь один объект этого класса.

---

*Не имеет значения, где находится ключевое слово virtual - до или после спецификатора ограничения доступа.



Если виртуальный базовый класс и производный класс разделяют имя какого-либо поля, фукции-члена или перечисления, то имя в производном классе скрывает имя в виртуальном базовом классе.

```
struct V { int i; void f(); /* ... */ };
struct A : virtual V { int i; int f(); /* ... */ };
struct B : virtual V { /* ... */ };

class C : public A, public B
{
  // ...
  void  g() { i = f(); }  // Правильно
  // Как A::i, так и A::f() скрывают V::i и V::f()
};
```

Их конструкторы, если таковые имеются, вызываются конструктором *последнего класса* в цепочке производных классов.

```
V::V( int n ) : i( n ) { /* ... */ }
A::A( int i ) : V( i ) { /* ... */ }
B::B( int i ) : V( i ) { /* ... */ }
C::C( int i ) : V( i ), A( i ), B( i ) { /* ... */ }
```

- В объявлении класса могут быть смешаны с невиртуальными базовыми классами.

- Могут вызывать нежелательные множественные вызовы функций-членов в виртуальном базовом классе.

```
class V
{
  // ...
public:
  void f() { /* ... */ }
  // ...
};

class A : virtual public V
{
  // ...
public:
  void f() { /* ... */ V::f(); }
  // ...
};

class B : virtual public V
{
  // ...
public:
  void f() { /* ... */ V::f(); }
  // ...
};

class C : public A, public B
{
  // ...
public:
  void f() { /* ... */ A::f(); B::f(); }
  // Заметьте, V::f() вызывается дважды
};
```

Для преодоления указанного эффекта можно опреде-

лить функцию real_f(), которая выполняет действия, специфичные для функции f() данного класса. Тогда функция f() будет выполнять последовательные вызовы сначала функции real_f(), а затем - функции f().

```
class V
{
  // ...
protected:
  void real_f() { /* ... */ }
  // ...
public:
  void f() { real_f(); }
  // ...
};

class A : virtual public V
{
  // ...
protected:
  void real_f() { /* ... */ }
  // ...
public:
  void f() { real_f(); V::f(); }
  // ...
};

class B : virtual public V
{
  // ...
protected:
  void real_f() { /* ... */ }
  // ...
public:
  void f() { real_f(); V::f(); }
  // ...
```

```
};

class C : public A, public B
{
  // ...
protected:
  void real_f() { /* ... */ }
  // ...
public:
  void f() { real_f(); A::real_f(); B::real_f(); V::real_f(); }
  // ...
};
```

Шаблоны...

- Расширяют понятие функции и класса, предоставляя средства их *параметризации*, то есть объявления функций и классов в терминах "любого типа"*.

Шаблоны функций...

- Это объявление функции, предваряемое *спецификацией шаблона*. Спецификация шаблона состоит из ключевого слова template, за которым следует список параметров, заключенный в угловые скобки < >.

- Имеют параметры типа, которые обозначаются ключевым словом class, за которым следует идентификатор**. Идентификатор служит для замещения имени типа. Может быть более одного параметра типа.

- Автоматически расширяются транслятором до полного описания функции так, как это необходимо.

---

*Шаблоны поддерживаются, начиная с версии C++ USL 3.0 (*Прим. пер.* Компилятор Borland C++ поддерживает шаблоны, начиная с версии 3.0).

**Ключевое слово class в контексте шаблонов означает *любой тип*, а не только класс.

```
// Обьявление: максимум из двух значений типа T
template <class T> const T& Max( const T& , const T& );
// ...
  template <class T> const T&  // Определение*
Max( const T& a, const T& b )
{
  return a > b ? a : b;
}
// ...
  int i, j;
  float a, b;
  // ...
  int    k = Max( i, j );  // Вызов Max( int , int )
  float  c = Max( a, b );  // Вызов Max( float , float )
```

- Могут быть перегружены другими функциями-шаблонами или обычными функциями.

```
template<class T> const T& Max( const T&, const T& );
template<class T> const T& Max( const T*, int );
int Max( int ( * ) ( int ), int ( * ) ( int ) );
```

- Для определенных типов могут быть перекрыты для того, чтобы выполнять (или не выполнять) какие-либо действия, которые функции-шаблоны не выполняют (или выполняют).

```
const char * & Max( const char * & c, const char * & d )
{
  // ...выполнить что-либо, специфичное для char * ...
}
```

*Если функция-шаблон в рассматриваемом примере должна работать с каким-либо классом, то в этом классе операция должна быть перегружена. В противном случае при связывании программы произойдет ошибка.

Шаблоны классов...

- Это обявления классов, предваряемые спецификацией template.

- Автоматически расширяются компилятором до полных определений классов так, как это необходимо.

- Не могут быть вложены в другие классы (в отличие от обычных классов).

```
// Обьявить класс Stack, который представляет собой
// стек для любых типов
template<class T> class Stack
{
  T  * v;  // Указатель на некоторый тип T
  int size, top;
public:
  Stack( int ezis );
  ~Stack();
  void Push( const T& );  // Поместить T в стек
  T&  Pop();  // Извлечь T из стека
  T&  Top() const;  // И т.д.
};

Stack <int> i;  // Стек для int
Stack<char*> cp;  // Стек для char*
```

- Могут иметь нетипированные (или только нетипированные) параметры. Значения, указанные для этих параметров, должны быть константными выражениями.

```
// Передать размер как параметр шаблона
template<class T, int size> class Stack
{
```

```
  T v[ size ];  // Массив элементов типа T*
  int top;
public:
  Stack() : top( -1 ) { }
  // ...
};

Stack<int, 20> tiny;
Stack<int, 500> huge;
```

Хотя стеки tiny и huge и хранят тип int, но все же это различные типы, поскольку они имеют разный размер стека. Это можно проиллюстрировать тем, что указатель Stack<int, 20> - это *не то же* самое, что указатель на Stack<int, 500>.

```
Stack<int, 20> * is20p = &tiny;  // Правильно
Stack<int, 500> * is500p = &tiny;  // Ошибка
```

- Могут быть порождены как от нешаблонных классов, так и от классов-шаблонов. А также могут порождать как нешаблонные классы, так и классы-шаблоны**.

```
class A { /* ... */ };
template<class T> class B : public A { /* ... */ };
template<class T> class C : public B { /* ... */ };
class D : public C<int>  { /* ... */ };
```

- Для определенных типов могут быть перекрыты для то-

---

*В этом случае преимущество использования нетипированных параметров состоит в том, что поле v создается без использования операции new (которая может выполниться неудачно).

**Когда от класса-шаблона порождается нешаблонный класс, всем параметрам класса-шаблона должны быть присвоены некоторые "реальные" значения. В примере это int.

го, чтобы выполнять (или не выполнять) какие-либо действия, которые шаблоны классов не выполняют (или выполняют).

```
// Объявим свой собственный стек для char*
class Stack<char*>
{
  char * * v;  // Указатель на char*
  int size, top;
public:
  Stack( int ezis );
  ~Stack();
  void Push( const char*& );
  char* & Pop();
  char* & Top() const;
};
```

- Могут также быть классами-структурами или классами-объединениями (см. "Классы-структуры и классы-объединения" в разделе "Классы").

Статические данные-члены...

- Разделяются всеми объектами класса для *каждого конкретного экземпляра* класса-шаблона.

- Определяются в области видимости файла (как и все статические поля), когда определение предваряется спецификацией template.

```
template<class T> class C
{
  static int i;  // Обычное статическое поле
  static T t;  // Параметризованное
  // ...
```

```
};

template<class T> int C<T>::i;  // Определить в
template<class T> T C<T>::t;  // области видимости файла
// ...
C<char> c;  // Имеет int C::i и char C::t
C<float> f;  // Имеет int C::i и float C::t
```

Шаблоны функций-членов...

- Определяются вне объявлений классов, к которым они принадлежат, при помощи спецификации template.

```
template<class T> void
   Stack<T>::Push( const T& element )
{
  if( top == size - 1 )
    error( "stack overflow" );
  else  v[++top] = element;
}
```

- Для определенных типов могут быть перекрыты для того, чтобы выполнять (или не выполнять) какие-либо действия, которые шаблоны функций-членов не выполняют (или выполняют).

```
void Stack<char*>::Push( const char*& ccpr )
{
  // Выполнить нечто особенное с char*
}
```

Дружественные функции...

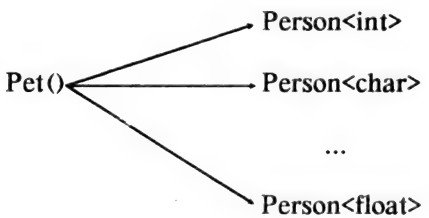
- Для каждого типа T могут быть друзьями всех классов типа T. Это обычные дружественные функции.

- Для типа T могут быть друзьями класса типа T.

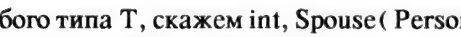
- Могут предваряться спецификацией template. Для типов T и U, функции-шаблоны типа U являются дружественными функциями каждому классу типа T.

```
template <class T> class Person
{
  friend void Pet();
  friend void Spouse( Person& );
  template<class U> friend void Coworker( U& );
};
// ...
void Pet() { /* ... */ }  // Обычная функция

template<class T> void Spouse( Person& p ) { /* ... */ }

template<class U> void Coworker( U& u ) { /* ... */ }
```

Здесь Pet() - функция, дружественная Person<T> для каждого типа T.



Для любого типа T, скажем int, Spouse(Person<int>&) - функция, дружественная Person<int>, но не Person<char>, или любому другому типу.

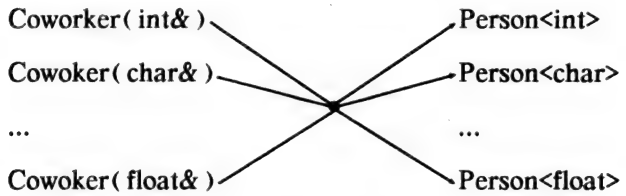


Spouse(Person<char>&) ——————→ Person<char>

... ...

Spouse(Person<float>&) ——————→ Person<float>

Coworker(U&) - функция, дружественная Person<T>, для любого типа T и любого типа U.



- Могут быть функциями-членами другого класса.

```
template<class T> class Person
{
  // ...
  friend void Family::Sibilding();
  friend void Acquaintance::Casual( Person& );
  template<class U> friend void
    Neighbor<U>::NextDoor( U& );
};
```

Здесь Family::Sibilding() - функция, дружественная Person<T>, для каждого типа T.

Для любого типа T, скажем для int, Acquaintance<int>::Casual(Person<int>&) - функция, дружественная Person<int>, но не Person<char>, или любому другому типу.

Neighbor<U>::NextDoor(U&) - функция, дружественная Person<T>, для любого типа U.

- Могут быть объявлены для всего класса.

```
template<class T> class Person
{
  // ...
  friend class Family;
  friend class Acquaintance;
  template<class U> friend class Neighbor;
};
```

Здесь для каждого типа T все функции-члены класса Family - это функции, дружественные Person<T>.

Для любого типа, скажем для int, все функции-члены класса Acquaintance<int> являются друзьями Person<int>, но не Person<char>, или любого другого типа.

Для каждого типа T и каждого типа U все функции-члены Neighbor<U> являются друзьями Person<T>.

- Могут также быть друзьями нешаблонных классов.

Перегрузка операций...

- Позволяет расширять на типы классов* значение любых операций, включая ::, sizeof, ? :, ., и .*.

- Осуществляется путем объявления функции, состоящей из ключевого слова operator, за которым следует одна из встроенных операций**.

---

*Не позволяют изменять приоритеты операций, ассоциативность или число операндов. Невозможно также изменить семантику операций для встроенных типов.

**Вы можете создавать новые операции.

- Не делает никаких предположений о подобных операциях. Например, если i имеет тип int, то ++i - это то же, что и i+=1, что то же самое, что и i=i+1. Для перегруженных операторов такие правила не выполняются до тех пор, пока пользователь сам не определит их.

- Не может использовать аргументы по умолчанию.

Унарные операторы...

- Могут быть объявлены как нестатические функции-члены, не имеющие аргументов. То есть @x интерпретируется как x.operator@() для любого оператора @.

- Могут быть объявлены как функции, не являющиеся членами какого-либо класса, которые имеют один аргумент. Этот аргумент является переменной данного класса либо ссылкой на такую переменную. В этом случае @x интерпретируется как operator@(x)* для любого оператора @.

```
class X
{
  X operator-() const;  // Унарный минус
  X operator&() const;  // Вычисление адреса
 X operator^() const;  // Ошибка: операция ^ бинарная
};

class Y
{
  friend Y operator-( const Y& );  // Унарный минус
  friend Y operator&( const Y& );  // Вычисление адреса
  friend Y operator^( const Y& );  // Ошибка: операция
                                   // ^ бинарная
```

*Делать ли унарные операции членами класса решает программист, хотя обычно поступяют именно так.

```
};
```

Бинарные операции...

- Могут быть объявлены как нестатические функции, имеющие один аргумент. То есть x@y интерпретируется как x.operator@(y) для любого оператора @.

- Могут быть объявлены как функции, не являющиеся членами какого-либо класса, имеющие один аргумент. Этот аргумент должен являться либо переменной данного класса, либо ссылкой на такую переменную*. То есть для любой операции @, включая =, x@y интерпретируется как operator@(x , y).

```
class X
{
  X operator-( const X& ) const;  // Бинарный минус
  X operator&( const X& ) const;  // Побитовое И
  X operator!( const X& ) const;  // Ошибка: ! унарная
                                  // операция
};

class Y
{
  friend Y operator-( const Y&, const Y& );
  friend Y operator&( const Y&, const Y& );
  friend Y operator!( const Y&, const Y& );  // Ошибка
};
```

Операция вызова функции...

- Должена быть объявлена как нестатическая функция

*Бинарные операторы чаще всего объявляются как дружественные функции. Это позволяет использовать в выражениях переменные класса и делать перегруженные операции коммутативными.

член-класса.

- Позволяет пользователю определять число операндов.

```
class X
{
  // ...
public:
  X( int a, int b, int c );
  // ...
  void operator () ( int i, int  j, int k );
};

X Xample( 1, 2, 3 );  // Вызывается конструктор
Xample( 4, 5, 6 );  // Вызывается операция ()
```

Операция присваивания...

- Используется для присвоения значения одного объекта данного класса другому.

- Если не определена пользователем, то выполняет присваивание значений полей одного объекта данного класса другому.

- Должна быть определена, когда класс содержит поля, которые являются указателями на динамически выделенную память*.

- Это только функция операции, которая *не* наследуется.

*Если операция присваивания не перегружена, то в приводимом примере выполнение операции присваивания по умолчанию приведет к тому, что указатель объекта b будет указывать на ту же область памяти, что и указатель объекта a. По той же причине должен быть определен и конструктор копирования (см. "Конструкторы" в разделе "Классы"). Если необходимо запретить операцию присваивания для объектов данного класса, то ее нужно просто *объявить* в закрытой части объявления класса.

- Должна быть определена как нестатическая функция-член класса.

```
class IntStack
{
  int  * v, size, top;
public:
  // ...
  IntStack& operator=( const  IntStack& );
};

IntStack& IntStack::operator=( const IntStack& from )
{
  if( this != &from )  // Проверка на from = from
  {
    delete[ ] v;
    v = new int[ size = from.size ];
    for( register int i = 0; i < size; ++i )
        v[i] = from.v[i];
    top = from.top;
  }
   return *this;  // Позволяет выполнять множественное
                  // присваивание
}
// ...
IntStack a( 100 ), b, c;
// ...
c = b = a;  // Присвоить значение переменной a
            // переменным b и c
```

Операция индексации...

- Должна быть определена как нестатическая функция-член класса.

- Чаще всего возвращает ссылку, что позволяет использовать ее в операции присваивания с обеих сторон.

```
class String*
{
  char * s;
  // ...
public:
  String( char );
  String( const char * );
  char& operator[ ]( int pos ) { return s[ pos ]; }
  // ...
};
String ball = "mitten";
ball[0] = 'k';  // char& допускает присваивание
```

Операция доступа к члену **...

- Для x->m интерпретируется как (x.operator->())->m***. Заметим, что это унарная операция, и что x - это объект класса, а *не* указатель на него.

- Должна возвращать указатель на объект класса, либо объект класса, либо ссылку на объект класса, для которого эта операция определена, поскольку оригинальное значение опреации -> не теряется, а только задерживается.

- Должна быть определена как нестатическая функция-член класса.

---

*В реальном определении класса String значение pos, передаваемое в operator[], должно проверяться на допустимость.

**Ранние версии C++ не позволяют перегружать эту операцию.

***Это неприменимо к бинарному оператору ->*, который не является особым.

Операции инкремента и декремента...

- Могут быть перегружены как для префиксной, так и для постфиксной формы записи*.

  Префиксная: Объявляется как операция-член класса, которая не имеет аргументов, либо как дружественная операция, которая имеет один аргумент, представляющий собой ссылку на объект данного класса.

  Постфиксная: Объявляется как операция-член класса, которая имеет один аргумент типа int, либо как дружественная операция, которая имеет два аргумента: ссылку на объект данного класса и аргумент типа int**.

```
class X
{
  //...
public:
  X& operator++(); // Префиксная форма
  X& operator++( int ); // Постфиксная форма
};

class Y
{
  //...
public:
friend Y& operator--( Y& ); // Префиксная форма
friend Y& operator--( Y&, int ); // Постфиксная форма
};
```

*Ранние версии C++ не имеют механизма разделения префиксной и постфиксной форм записи этой операции.

**Аргумент int на самом деле не используется. Фактически он имеет нулевое значение.

Операции new и delete...

- Предоставляют классу механизм управления собственной памятью*.
- Должны иметь тип возвращаемого значения void* для new и void для delete.
- Требуют параметр типа size_t, который определен в стандартном файле заголовка <stddef.h>.
- Неявно являются статическими функциями-членами и, следовательно, не могут быть ни константными, ни виртуальными.

```
#include <stddef.h>

class Thing
{
  // ...
  enum { block = 20 };
  static Thing   * freeList;
public:
  // ...
  void * operator new( size_t );
  void   operator delete( void *, size_t );
};

void *Thing::operator new( size_t size )
{
  Thing  * p;
  if( !freeList )
  {
    freeList = p = ( Thing * )new char[size*block];
```

*Ранние версии C++ не позволяют перегружать операторы new и delete.

```
    while( p != &freeList[ block-1 ])
            p->next = p+1, ++p;
    p->next = 0;
  }
  p=freeList, freeList = freeList->next;
  return p;
}

void Thing::operator delete( void * p, size_t )
{
  ( (Thing *)p )->next = freeList;
  freeList = (Thing *)p;
}
```

- Если определены для класса, *не* могут быть вызваны при выделении и освобождении памяти для массивов объектов данного класса.

- Операция new может быть перегружена, чтобы принимать дополнительные аргументы. Передаваемые аргументы, заключенные в скобки, помещаются при вызове после ключевого слова new.

```
class Thing
{
  // ...
public:
  // ...
/*
** Добавить дополнительный аргумент buf, который
** будет использоваться для выделения памяти для
** Thing по указанному адресу
*/
void * operator new( size_t size, void * buf );
};
//...
```

```
void * Thing::new( size_t size, void * buf )
{
  // Разместить Thing по адресу buf
}
// ...
char buffer[1000];
Thing  * p = new( buffer ) Thing;
```

Перегруженная операция new с дополнительными аргументами скрывает глобальную операцию ::operator new. Теперь при обращении к new с обычным синтаксисом произойдет ошибка.

```
Thing *p = new Thing;  // Ошибка: отсутствует аргумент
```

Чтобы снова использовать обычный синтаксис оператора new, для класса должна быть предусмотрена функция-член new, имеющая только один аргумент - size_t size. Она явно вызывает глобальную операцию ::operator new.

```
class Thing
{
  // ...
  void * operator new( size_t size, void * buf );
  void * operator new( size_t size )
  {
    return ::operator new( size );
  }
};
```

Препроцессор

В этом разделе описан препроцессор C++. Хотя он, строго говоря, и не является частью собственно языка, все программы используют его.

Препроцессор C++ базируется на препроцессоре ANSI C. В многих реализациях C++, однако, используется препроцессор ANSI C или даже "классического C". При этом могут возникать определенные проблемы с комментариями.

Строки, в которых символ # является первым непустым символом, являются директивами препроцессора. Строки, содержащие директивы, заканчиваются символом новой строки. Обратная косая (\) используется для продолжения директивы на новой строке.

Комментарии...

- Как упоминалось ранее, могут вызывать определенные проблемы при работе со старыми препроцессорами, поскольку они не распознают комментарии C++.

```
#define SIZE256  // Размер чего-нибудь
// ...
  for( int i = 0; i < SIZE; ++i )
    // ...
// После обработки препроцессором не C++

for( int i = 0; i < 256  // Размер чего-нибудь; ++i )
 // Скорее всего в этой точке возникнет
 // синтаксическая ошибка
```

Комментарий, начинающийся с символа //, вместо того, чтобы быть удаленным, считается продолжением директивы препроцессора.

Во избежание подобных ошибок в строках, содержащих директивы препроцессора, рекомендуется использовать комментарии, используя символы /* ... */.

```
#define SIZE 256  /* Размер чего-нибудь */

for( int i = 0; i < 256; ++i )
  // Как и предполагалось
```

Предопределенные имена...

- Содержат различную информацию о процессе компиляции.

__cplusplus	Определено для всех реализаций C++.
__LINE__	Текущий номер строки исходного файла.
__FILE__	Имя текущего исходного файла.
__DATE__	Текущая дата: Mmm dd yyyy.
__TIME__	Текущее время: hh:mm:ss.

Пустая директива...

- Состоит из отдельного символа #.
- Не дает никакого эффекта.

#line константа "имя_файла"

- Находя подобную строку, компилятор считает, что *константа* и *имя_файла* задают номер следующей компи-

лируемой строки и имя входного файла*.

Константа устанавливает номер текущей строки (__LINE__), а *имя_файла* - имя текущего файла (__FILE__). Имя файла является необязательным параметром.

#include <имя_файла>
#include "имя_файла"

- Включает полный текст файла в текущий файл**.

При использовании директивы в первой форме поиск включаемого файла будет осуществляться в стандартных для данной реализации C++ директориях. При использовании второй формы поиск осуществляется в текущем директории.

```
#include <iostream.h> // Используется стандартный
                      // директорий
#include "MeHeader.h" // Используется текущий
                      // директорий
```

#pragma текст

- Вызывает некоторые зависящие от реализации действия. Нераспознаваемый *текст* игнорируется.

---

*Этот механизм используется в основном системными программами (такими, как компилятор C++).

**Включаемые файлы, в свою очередь, могут включать другие файлы. Глубина вложенности таких включений определяется конкретной реализацией.

#define идентификатор текст
#define идентификатор (идентификатор , ...) текст

- Первая форма связывает *идентификатор* с *текстом* так, что когда *идентификатор* встречается в программе, он заменяется на *текст*[*].

```
#define forever for( ; ; )
// ...
 forever { /* Вечный цикл */ }
```

- Во второй форме допускается использование аргументов[**]. Между первым *идентификатором* и скобкой пробел недопустим.

  Для обеспечения правильной подстановки идентификаторы в *тексте* должны быть заключены в скобки.

```
#define MIN(a,b) ( (a)  (b) ? (a) : (b) )
```

  Однако *идентификаторы* в *тексте* не будут подставляться, если они заключены в кавычки. Для отказа от этого используется операция #.

```
#define MAIL(logid) "/usr/mail/" #logit
// ...
if stream mailFile( MAIL( pjl ) );
// MAIL( pjl ) становтися "/usr/mail/" "pjl", которые
// затем соединяются в "/usr/mail/pjl"
```

[*] Является устаревшим дополнением к объявлению const.

[**] Является устаревшим дополнением к объявлению inline.

#undef идентификатор

- Разрывает связь между *идентификатором* и текстом, с которым он был связан.

```
#undef MIN
```

#error текст

- Вызывает появление сообщения, содержащего *текст*.

Условная компиляция...

- Вызывает деление текста программы на части, которые компилируются (или не компилируются) в зависимости от некоторых внешних условий.

- Начинается в одном из трех случаев:

```
#if константное-выражение        /* Истинно? */

#ifdef идентификатор             /* Определен? */

#ifndef идентификатор            /* Не определен? */
```

Для #if вычисляется *константное-выражение*, и если результат вычислений - истина (ненулевой), то компилируются все строки программы вплоть до директивы #else, #elif или #endif.

Для #ifdef (и #ifndef) компилируется соответствующая часть кода, если определен (не определен) *идентификатор*.

В *константном-выражении* может быть использована операция defined для проверки, был ли определен указанный *идентификатор*. Обратное условие проверяет

операция !defined*.

- Может иметь раздел "иначе", определяемый в одной из двух форм:

  #else

  #elif *константное выражение*

  Для #else соответствующая последовательность кода компилируется в том случае, если не выполняется условие, указанное в #if.

  #elif аналогично #else за исключением того, что должно еще выполняться условие, задаваемое *константным-выражением*.

- Должна заканчиваться дирктивой #endif.

```
#ifdef DEBUG
cerr << "counter = "<< counter << endl;
#endif

#if defined SYSV
  extern long f();
#elif defined BSD
  extern short f();
#else
  extern int f();
#endif
```

- Часто используется для предупреждения повторного включения файлов заголовков (".h").

*Ранние версии препроцессора не имеют операции defined, а работают только с #ifdef и #ifndef. Эти директивы были сохранены для совместимости (кроме того, запись #ifdef короче чем #if defined).

```
// Phone.h

#ifndef __PHONE__
#define __PHONE__

class Phone { /* ... */ };

#endif

// PayPhone.h

#ifndef __PAYPHONE__
#define __PAYPHONE__

#include "Phone.h"

class PayPhone : public Phone { /* ... */ };

#endif

// my.c
#include "Phone.h"
#include "PayPhone.h"
//...
```

В файле my.c используется класс Phone, определенный в файле Phone.h, и класс PayPhone, определенный в файле PayPhone.h.

Первая директива #include в файле my.c включает в него файл Phone.h. При этом определяется идентификатор __PHONE__ и происходит объявление класса Phone.

Затем вторая директива #include включает в файл my.c

файл PayPhone.h. При этом определяется идентификатор __PAYPHONE__. Поскольку класс PayPhone порожден от класса Phone, в файл PayPhone.h включается файл Phone.h. Но поскольку идентификатор __PHONE__ уже определен, повторного объявления класса Phone не происходит.

Если не использовать этот механизм, то произойдет повторное объявление класса Phone.

Библиотека ввода/вывода

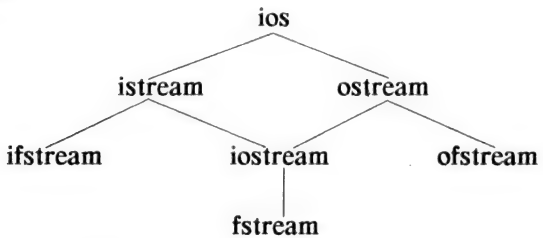
В C++ отсутствуют встроенные средства ввода/вывода. Ввод/вывод осуществляется через библиотеку классов*. Для того, чтобы получить к ней доступ, необходимо включить в программу соответствующие файлы заголовков**.

Классы

- В библиотеке ввода/вывода имеются следующие классы:

ios	Базовый потоковый класс
istream	Класс потока ввода
ostream	Класс потока вывода
iostream	Класс потока ввода и вывода
ifstream	Класс файла ввода
ofstream	Класс файла вывода
fstream	Класс файла ввода и вывода

- Их связи в терминах производных классов представлены ниже:



*Поддерживается также библиотека stdio из C. Однако лучше использовать новые библиотеки C++.

**Различные реализации поддерживают не все файлы заголовков.

Состояния ошибки...

- Пдоддерживаются каждым из потоков.
- Хранятся как набор битов: eofbit, failbit и badbit. Множественные состояния получаются при помощи операции побитового ИЛИ.

int eof()

- Возвращает ненулевое значение, если в потоке обнаружен символ конца файла.

int bad()

- Возвращает ненулевое значение, если при выполнении некоторой операции над потоком произошла ошибка. Восстановление в этом случае маловероятно.

int fail()

- Возвращает ненулевое значение, если при выполнении некоторой операции, такой как извлечение или преобразование, в потоке произошла ошибка. При этом состоянии возможно восстановление, а поток может использоваться дальше, поскольку при восстановлении состояние ошибки сбрасывается. Кроме того, возвращает ненулевое значение, если ненулевое значение возвращает bad() (но при этом восстановление маловероятно).

int good()

- Возвращает ненулевое значение, если eof(), bat() и fail() возвращают ноль.

int rdstate()

- Возвращает текущее состояние ошибки.

void clear(int state)

- Устанавливает состояние ошибки потока.

```
istream s;
// Устанавливает failbit, не трогая другие
s.clear( ios::failbit | s.state() );
```

int operator!()
operator void*()

- Первая возвращает ненулевое значение, если установлен failbit или badbit. Вторая возвращает нулевое значение, если установлен failbit или badbit.

```
if( !myStream ) // operator!()
if( myStream ) // operator void*()
```

Потоковый вывод...

- Достигается при использовании переменных классов ostream или iostream.

- Может применяться с одним из предопределенных выходных потоков или с выходным потоком, который был определен пользователем. Имеются следующие предопределенные выходные потоки:

cout	ostream, связанный со стандартным

	устройством вывода;
cerr	ostream, связанный со стандартным устройством сообщения об ошибках.

Операция вставки...

- Записывает последовательность символов в указанный поток.
- Это перегруженная* операция <<.
- Предопределена для всех простых типов и указателей на char и void.

```
cout << "Wow! This is neat!/n";
cerr << "Bad operand: " << op << '/n';
```

- Может быть перегружена для классов**.

```
class Phone
{
  int area, exchange, line;
  // ...
  friend ostream& operator<<(ostream&, const Phone& );
};

ostream& operator<<( ostream& os, const Phone& p )
{
  os << '(' << p.area << ") " << p.exchange << '-' << p.line;
  return os;  // Должна возвращать ostream&
```

*При использовании операции вставки обращайте внимание на ее приоритет. Операции с более низким приоритетом должны заключаться в скобки.

**При необходимости доступа к членам класса, находящимся в закрытой части, операцию вставки объявляют как дружественную.

```
}
// ...
cout << "Home number: " << home << '/n';
```

ostream& put(char)

- Записывает отдельный символ в поток*.

```
cout.put( '/n' );  // То же, что и cout << '/n';
cout.put( '!' ).put( bell );
```

ostream& write(
const char* buf, int count)
ostream& write(
const unsigned char* buf, int count)

- Записывает в поток указанное число символов из буфера, на который указывает buf.

```
const int bufSize = 2048;
char buf[ bufSize ];
// ...
cout.write( buf, sizeof buf );
```

ostream& flush()
ostream& flush(ostream&)

- Вызывает запись содержимого потока. Первая функция - это функция-член класса, а вторая - манипулятор.

---

*Выбор между этим методом и операцией >> определяется только вкусом программиста.

```
cout << "What's your name?/n"
cout.flush();
// ...
cout << "Where do you live?/n" << flush;
```

ostream& seekp(streampos, seek_dir = ios::beg)

- Перемещает позицию указателя записи в файле. streampos - это значение целого типа, а seek_dir - это точка отсчета, от которой отсчитывается значение новой позиции: начало файла, текущая позиция или конец файла.

```
fileA.seekp( i*100 );  // i блоков по 100
fileB.seekp( 100, ios::cur );   // Вперед на 100 байт
fileC.seekp( 0, ios::end );  // Последний символ
```

streampos tellp()

- Возвращает текущую позицию указателя записи в файле потока в байтах. streampos - это значение целого типа.

```
streampos mark = myFile.tellp();
```

ostream& endl(ostream&)

- Вставляет символ '\n' и выполняет flush.

```
cout << "Hello, world!" << endl;
```

Потоковый ввод...

- Достигается при использовании переменной класса istream или iostream.
- Может применяться с предопределенным входным потоком cin или с входным потоком, определенным пользователем.

Операция извлечения...

- Читает последовательность символов из указанного потока. Промежутки между символами удаляются*.
- Это перегруженная операция >>.
- Предопределена для всех простых типов и указателя на char.

```
int age, weight;
cout << "Enter age and weight: " << flush;
cin >> age >> weight;   // Считывает возраст, а затем
                        // вес
```

- Имеет ненулевое значение до тех пор, пока не встретит символ конца файла (используя операцию преобразования operator void*()).

```
char c;
while( cin >> c ) //Копировать непустые символы из cin
                  // в cout
    cout << c;
```

*Для чтения символов пустых промежутков используйте методы get(), getline() или read().

- Устанавливает failbit, если символы, считываемые из потока, не имеют требуемый тип. Устанавливает badbit, если при чтении произошла ошибка.

- Для классов может быть перегружена тем же способом, что и операция вставки (см. "Операция вставки" в разделе "Потоковый вывод").

int get()

- Извлекает и возвращает один символ из входного потока, включая символы пустых промежутков (прим. переводчика: пробел, табуляция и т.п.).

- Возвращает значение EOF, если достигнут конец файла* (никогда не устанавливает failbit).

```
int c;
while( ( c = cin.get() ) != EOF )
  cout.put( c );
```

istream& get(char&)
istream& get(unsigned char&)

- Извлекает и возвращает один символ из входного потока, включая символы пустых промежутков.

- Имеет ненулевое значение до тех пор, пока не будет достигнут конец файла.

*Значение EOF должно отличаться от всех остальных символов. Поэтому его значение как правило равно -1. В приводимом примере переменная с не может быть объявлена как char, поскольку компилятор может считать char как unsigned char, поэтому с объявлена как int.

```
char c;
while( cin.get( c ) ) cout.put( c );
```

```
istream& get( char* buf,
              int limit, char delim = '\n' )
istream& get( unsigned char* buf,
              int limit, char delim = '\n' )
```

- Читает символы из потока до тех пор, пока не будет считано указанное количество символов либо пока не будет считан символ-ограничитель. Ограничитель не включается в считанную последовательность символов и остается в потоке.

```
istream& getline( char* buf, int limit,
                  char delim = '\n' )
```

- Читает символы из потока до тех пор, пока не будет считано указанное количество символов, либо пока не будет считан символ-ограничитель. Ограничитель не включается в считанную последовательность символов.

```
istream& read( char * buf, int count )
istream& read( unsigned char* buf,
               int count )
```

- Читает строку символов из потока.

```
const int bufSize = 2048;
char buf[ bufSize ];
// ...
```

```
cout.read( buf, bufSize );
```

- Если конец файла достигнут до того, как было считано указанное число символов, то устанавливается failbit.

istream& ignore(int limit=1, int delim=EOF)

- Удаляет из потока указанное число символов, пока не встретится символ-ограничитель.

int gcount()

- Возвращает число символов, прочитанных из потока во время выполнения последней операции неформатированного ввода.

istream& putback(char)

- Помещает символ обратно в поток. Как правило, между успешными вызовами get() в поток может быть возвращен один символ. Символ, помещаемый обратно в поток, должен быть символом, который был считан из потока, в противном случае результат непредсказуем.

int peek()

- Возвращает следующий символ, который будет считан из потока, фактически не извлекая его из потока.

istream& seekg(streampos, seek_dir=ios::beg)

- Перемещает позицию указателя чтения в файле. streampos - это значение целого типа, а seek_dir - это точка отсчета, от которой отсчитывается значение новой

позиции: начало файла, текущая позиция или конец файла.


```
fileA.seekg( i*100 );  // i блоков по 100
fileB.seekg( 100, ios::cur );  // Вперед на 100 байт
fileC.seekg( 0, ios::end );  // Последний символ
```

streampos tellg()

- Возвращает текущую позицию указателя чтения в файле потока в байтах. streampos - это значение целого типа.


```
streampos mark = myFile.tellg();
```

istream& ws(istream&)

- Извлекает и удаляет символы пустых промежутков из указанного потока.


```
cin >> ws;  // Удаляет лидирующие пробелы
// ...
```

Файловый вывод...

- Достигается при связывании выходного потока с файлом при помощи переменной класса ofstream или fstream. Эти классы определены в файле заголовка fstream.h.

- Может использоваться точно так же, как и обычный потоковый вывод, поскольку класс ofstream порожден от класса ostream.

- Должен осуществлять проверку ошибок.

```
ofstream outFile( "myFile" );  // Попытаться открыть
                               // myFile
if( !outFile )  // Открытие невозможно
{
  cerr << "cannot open /"myFile/" for output/n";
  exit( 1 );
}
outFile << "Wow! This is even neater!/n";
```

- Уничтожает все данные, хранившиеся в файле до его открытия, если только не установлен режим добавления (см. список режимов в пункте "Открытие" раздела "Остальной файловый ввод/вывод").

```
ofstream afterthought( "myFile", ios::app );
afterthought << "On the other hand.../n";
```

Файловый ввод...

- Достигается при связывании входного потока с файлом при помощи переменной класса ifstream или fstream. Эти классы определены в файле заголовка fstream.h.

- Может использоваться точно так же, как и обычный потоковый ввод, поскольку класс ifstream порожден от класса ostream.

- Должен осуществлять проверку ошибок.

```
ofstream inFile( "myFile" ); // Попытаться открыть
                             // myFile
if( !inFile ) // Открытие невозможно
{
  cerr << "cannot open /"myFile/" for output/n";
```

```
  exit( 1 );
}
int a, b;
inFile >> a >> b;
```

Остальной файловый ввод/вывод

```
void open( char * name,
           int mode = ios::out,
           int prot = filebuf::openprot )
```

- Открывает файл и связывает его с открытым ранее потоком. Параметр mode может иметь следующие значения*:

ios::app	Все записываемые данные добавляются в конец файла (подразумевается ios::out);
ios::ate	Все записываемые данные добавляются в конец файла (не подразумевается ios::out);
ios::in	Файл открывается для ввода;
ios::out	Файл открывается для вывода;
ios::trunc	Усечение предыдущего содержимого файла (подразумевается ios::out);
ios::nocreate	Если файл не существует, то открыть его невозможно;
ios::noreplace	Если файл уже существует, то открыть его невозможно.

*Режимы могут комбинироваться при помощи операции побитового ИЛИ.

- Устанавливает режим защиты prot*.

void close()

- Закрывает файл и открепляет его от потока.

```
ifstream encyclopaedia;
// ...
for( int i = 0; i < numVolumes; ++i )
{
  encyclopaedia.open( volume[ i ] );
  if( !encyclopaedia )
  // Обработка ошибки открытия файла
  // Обработка файла
  encyclopaedia.close();
}
```

Состояния форматирования...

- Управляют появлением чисел при выполнении операции вставки в выходной поток и форматированием при выполнении операции извлечения из входного потока.

- Устанавливаются при помощи различных флагов, которые могут быть изменены при помощи методов flags(), setf() и unsetf().

- Флаги форматирования могут быть установлены при помощи метода setf(). Существуют следующие флаги.

skipws	Если установлен этот флаг, то пустые промежутки при вводе посредством

*Зависит от операционной системы.

	операции >> будут опускаться.
left, right, internal	Управляют дополнением значения символом дополнения (см. ниже fill()) при левом или правом выравнивании*. Эти три бита составляют статическое поле ios::adjustfield.
dec, oct, hex	Управляют системой счисления, которая используется при выводе целых типов. По умолчанию для вставки выбирается десятичная система счисления, а для извлечения - форматы записи, используемые в C++ для записи целых констант. Эти биты составляют статическое поле ios::basefield.
showbase	Если установлен этот бит, то при выводе целых чисел указывается система счисления. Числа, начинающиеся с 0, - это восьмиричные числа, с 0x или 0X - шестнадцатиричные.
showpoint	Если установлен этот бит, то при отображении чисел с плавающей точкой выводится десятичная точка и хвостовые нули.
showpos	Если установлен этот бит, то положительные целые числа будут выводиться со знаком '+'.
scientific, fixed	Если установлен бит scientific, то при вставке значений с плавающей точкой будет использоваться так называемая

*По умолчанию используется выравнивание по правому краю. Когда установлен бит internal, символ-заполнитель вставляется между знаком или указателем системы счисления и значением.

научная запись*. Если установлен бит fixed, то после десятичной точки вставляется устанавливаемое методом precision() число цифр. Если ни один из этих битов не установлен, то числа с плавающей точкой вставляются в соответствии со следующим правилом: научная запись используется, когда значение экспоненты меньше -4 или больше текущей точности. Эти биты составляют статическое поле ios::floatfield.

uppercase — Если установлен этот бит, то при выводе шестнадцатиричных значений будет использоваться прописная X, а при выводе значений с плавающей точкой - прописная E.

long flags()

- Возвращает текущие флаги форматирования.

long flags(long)

- Устанавливает указаные флаги форматирования и возвращает их предыдущее значение.

```
long oldFlags = cout.flags();
// ...изменяются какие-то флаги...
cout.flags( oldFlags );
```

* То есть одна десятичная цифра перед запятой, десятичная точка, некоторое количество цифр, число которых задается методом precision(), символ e или E, в зависимости от состояния флага uppercase.

long setf(long bitFlags)

- Включает указанные флаги форматирования и возвращает их предыдущее значение.

```
long oldFlags = cout.setf( ios::showbase );
```

long setf(long bitFlags, long bitField)

- Очищает указанные битовые поля, затем устанавливает указанные флаги и возвращает их первоначальное значение*.

- Должна использоваться для битовых флагов ios::adjustfield, ios::basefield или ios::floatfield.

```
cout.setf( ios::scientific, ios::floatfield );
```

long unsetf(long)

- Выключает указанные флаги форматирования и возвращает их предыдущее значение.

```
cout.unsetf( ios::showbase & ios::uppercase );
```

ios& setiosflags(long)

- То же, что и потоковый метод flags(), за исключением того, что setiosflags() - это потоковый манипулятор**.

---

*Важное отличие между этой версией setf() и предыдущей состоит в том, что эта версия сначала очищает битовые поля. Предыдущая версия будет устанавливать ios::scientific, не проверяя, установлен ли бит ios::fixed.

**Для использования setiosflags() необходимо включить в программу файл заголовка iomanip.h.

```
cout << setiosflags( ios::showpos );
```

ios& resetiosflags(long)

- То же, что и потоковый метод unsetf(), за исключением того, что resetiosflags() - это потоковый манипулятор*.

```
cout << resetiosflags( ios::showpos );
```

char fill(char)

- Устанавливает символ-заполнитель и возвращает предыдущий символ-заполнитель. По умолчанию в качестве символа-заполнителя используется пробел (см. ниже width()).

ostream& setfill(char)

- То же, что и потоковый метод fill(), за исключением того, что setfill() - это потоковый манипулятор**.

char fill()

- Возвращает текущий символ-заполнитель.

int width(int minimum)

- Устанавливает минимальную ширину поля для данного размера и возвращает предыдущую ширину поля. Ноль означает отсутствие минимума.

- Когда при вставке в поток или извлечении из потока

---

*Для использования resetiosflags() необходимо включить в программу файл заголовка iomanip.h.

**Для использования setfill() необходимо включить в программу файл заголовка iomanip.h.

вставляемое или извлекаемое значение меньше минимальной ширины поля, то недостающие символы заполняются символом-заполнителем. Если число символов больше или равно минимуму, то ничего не происходит.

- Минимальная длина поля сбрасывается в ноль после каждой операции вставки или извлечения.

int width()

- Возвращает текущую минимальную длину поля.

ios& setw(int size)

- Предотвращает переполнение массива вводимых символов, обрывая строку, длина которой больше, чем указанный буфер*.

```
const int lineSize = 80;
char line[ lineSize ];
// ...
while( cin >> setw( lineSize ) >> line )
//...
```

ios& dec(ios&)
ios& oct(ios&)
ios& hex(ios&)
ios& setbase(int)

- Изменяют систему счисления, используемую для представления целых чисел при выполнении операций вставки и извлечения. setbase(int) - это потоковый

*Чтобы использовать setw(), необходимо включить в программу файл заголовка iomanip.h.

манипулятор*.

```
int n = 1991;
cout << "Decimal: " << n
<< oct << ", Octal: " << n
<< hex << ", Hexadecimal: " << n << endl;

// После выполнения получим...

Decimal: 1991, Octal: 3707, Hexdecimal: 7c7
```

```
int d, o, h;
cin >> d >> oct >> o >> hex >> h;
```

```
cout << setbase( 16 );
```

int precision(int)

- Устанавливает число значащих цифр, используемое при выводе чисел с плавающей точкой, и возвращает его предыдущее значение. По умолчанию принимается шесть цифр.

ios& setprecision(int)

- То же, что и потоковый метод precision(), за исключением того, что setprecision() - это потоковый манипулятор**.

---

*Чтобы использовать setbase(), необходимо включить в программу файл заголовка iomanip.h.

**Для использования setprecision() необходимо включить в программу файл заголовка iomanip.h.

int precision()

- Возвращает текущее значение числа значащих цифр, используемых для вывода чисел с плавающей точкой.

Другие библиотеки

Существует множество библиотек для C++ (библиотеки для C могут использоваться с C++). Те из них, которые стали общими бля большинства реализаций, приведены ниже.

ctype.h...

- Предоставляет макросы для проверки и обработки символов способом, не зависящим от реализации.

Следующие макросы имеют форму: int *макро*(char c)

isalnum	c - это буква или десятичная цифра
isalpha	c - это буква
isascii	c - это символ ASCII (с кодом меньше, чем 128)
iscntrl	c - это символ управления или удаления (127)
isdigit	c - это десятичная цифра
isgraph	c - это печатный символ, отличный от пробела
islower	c - это символ в нижнем регистре
isprint	c - это печатный (не управляющий) символ
ispunct	c - это знак препинания, то есть ни управляющий символ, ни алфавитно-цифровой символ, ни пробел
isspace	c - это символ пустого промежутка
isupper	c - это символ в верхнем регистре
isxdigit	c - это шестнадцатиричная цифра

Следующие макросы имеют форму: char *макро*(char c)

toascii	преобразует целое в символ ASCII
_tolower	преобразует символ из верхнего регистра в нижний
_toupper	преобразует символ из нижнего регистра в верхний
tolower	то же, что и (isupper(c) ? _tolower(c) : c)
toupper	то же, что и (islower(c) ? _toupper(c) : c)

string.h...

- Предоставляет функции для обработки строк, то есть последовательностей символов, которые заканчиваются нулем.

int strlen(const char * s)

- Возвращает длину строки s, то есть число символов перед нуль-терминатором.

```
char *message = "Hello world!";
int lenght = strlen( message );  // lenght = 12
```

char * strcpy(char * to, const char * from)
char * strncpy(char * to, const char * from, int limit)

- Копирует строку, на которую указывает from, в строку, на которую указывает to. Возвращает значение to. Область, на которую указывает to, должна иметь достаточ-

ный размер для размещения копируемой строки.

- strncpy() копирует не более, чем limit символов. Скопированная строка заканчивается нулевым символом.

```
char *message = "You whant it when?";
char buf[20];
strcpy( buf, message );  // Копирует message в buf
```

```
char * strcat( char * to,
               const char * from )
char * strncat( char * to,
                const char * from, int limit )
```

- Добавляет копию строки, на которую указывает from, в конец строки, на которую указывает to. Возвращает to. Область памяти, на которую указывает to, должна иметь достаточный размер для размещения полученной строки.

- strncat() добавляет не более, чем n символов. Результирующая строка заканчивается нулевым символом.

```
char name[20] = "Humpty";
strcat( name, " Dumpty" );
```

```
int strcmp( const char * s1,
            const char * s2 )
int strncmp( const char * s1,
             const char * s2, int limit )
```

- Лексикографически сравнивает две строки. Возвращает

-1, 0 или +1, если s1 соответственно меньше, равна или больше, чем s2.

- strncmp() сравнивает не более, чем limit символов.

```
for( int first = 0, last = N; first <= last; )
{
  int index = ( first + last ) / 2;
  int result = strcmp( word, dictionary[index] );
  if( result < 0 )  last = index - 1;
  else
   if( result > 0 )
     first = index + 1;
   else  break;
}
```

char * strchr(const char * s, char c)
char * strrchr(const char * s, char c)

- Отыскивает первое вхождение в строку s символа c. Возвращает указатель на символ c в строке s. В противном случае возвращает нулевой указатель.

- strrchr(), (обратная) отыскивает первое вхождение символа c в строку s.

char * strpbrk(const char * s,
const char * set)

- Ищет первое вхождение любого из символов, содержащихся в строке set, в строку s. В случае успешного поиска возвращает указатель на символ в строке s. В противном случае возвращает нулевой указатель.

```
int strspn( const char * s,
            const char * set )
int strcspn( const char * s,
             const char * set )
```

- Начиная с начала строки s, возвращает число символов, которые содержатся в set. Останавливается при появлении первого символа, который не содержится в set.

- strcspn() возвращает число символов, которые не входят в set. Останавливается при появлении первого символа, содержащегося в set.

```
char *s = "2000 North Naperville Road";
int n = strspn( s, "0123456789" );  // n = 4
```

```
char * strtok( const char * s,
               const char * set )
```

- Делит строки на лексемы, используя символы, содержащиеся в set, как ограничители. После каждого вызова возвращает указатель на начало лексемы, которая заканчивается нулевым символом, либо нулевой указатель, если лексем больше нет.

- При первом вызове strtok() ей передается адрес строки. При втором и последующих вызовах ей передается нулевой указатель. Это указывает функции на необходимость продолжения обработки данной строки.

- Когда strtok() находит лексему, обнаруживая разделитель, она записывает в строку вместо него нулевой сим-

вол для того, чтобы отделить данную лексему*. Между вызовами strtok() сохраняет указатель на следующий символ для того, чтобы продолжить обработку строки при следующем вызове.

```
void ToWords( register char *s )
{
  register char * word;
  while( word = strtok( s, " " ) )
  {
   cout << word << endl;
   s = 0;  // Для продолжения устанавливает в ноль
  }
}
```

limits.h...

- Предоставляет набор не зависящих от реализации констант (приводимый список значений - это тот минимум, который требует стандарт ANSI).

CHAR_BIT	8	Число битов в байте
SCHAR_MIN	-127	Минимальное значение для signed char
SCHAR_MAX	+12	Максимальное значение для signed char
CHAR_MIN	0 SCHAR_MIN	Минимальное значение для char
CHAR_MAX	UCHAR_MAX SCHAR_MAX	Максимальное значение для char
MB_LEN_MAX	1	Минимальное число байт в многобайтовом

*Если вы не хотите изменять строку, то сначала сделайте ее копию.

		символе
SHRT_MIN	-32767	Минимальное значение для short
SHRT_MAX	+32767	Максимальное значение для short
USHRT_MAX	65535	Максимальное значение для unsigned short
INT_MIN	-32767	Минимальное значение для int
INT_MAX	+32767	Максимальное значение для int
UINT_MAX	65535	Максимальное значение для unsigned int
LONG_MIN	-2147683647	Минимальное значение для long
LONG_MAX	+2147683647	Максимальное значение для long
ULONG_MAX	4294967295	Максимальное значение для unsigned long

math.h...

- Предоставляет тригонометрические и другие математические функции.

  Все тригонометрические функции (sin, cos и т.д.) используют радианы.

  Все функции, за исключением тех, которые указаны, принимают один аргумент типа double. Все функции возвращают double.

acos	Аркосинус
asin	Арксинус
atan	Арктангенс
atan2(y,x)	Арктангенс от y/x
ceil	Округление в большую сторону
cos	Косинус
cosh	Гиперболический косинус
exp	*e* в степени x
fabs	Абсолютное значение
floor	Округление в меньшую сторону
fmod(x,y)	Остаток от деления x на y
log	Натуральный логарифм
log10	Десятичный логарифм
pow(x,y)	x в степени y
sin	Синус
sinh	Гиперболический синус
sqrt	Квадратный корень
tan	Тангенс
tanh	Гиперболический тангенс

stdarg.h...

- Предоставляет механизм для написания функций, которые принимают варьирующееся количество аргументов неопределенного типа.

- Для этой цели определены три макроса:

  void va_start(va_list, *последний-аргумент*)
  void va_arg(va_list, *тип*)
  void va_end(va_list)

  va_start инициализирует список аргументов, где *последний-аргумент* - это последний аргумент, указываемый в объявлении.

  va_arg ищет следующий аргумент в списке аргументов

указанного типа.

va_end очищает список аргументов.

```
void NumPrint( const char * format ... )
{
  va_list arg;
  va_start( arg, format );
  for( char *p = format; *p; ++p )
  {
   if( *p == '%' )
     switch( *++p )
     {
     case 'd':
       int ival = va_arg( arg, int );
       cout << ival;
       break;
     case 'f':
       double dval = va_arg( arg, double );
       cout << dval;
       break;
     }
   else
     cout << *p;
  }
  va_end( arg );
}
```

stdlib.h...

- Предоставляет набор функций для выполнения различных задач, включая завершение программы и преобразование чисел.

void exit(int)

- Вызывает прекращение выполнения программы*.
- Возвращает код завершения программы в вызвавшее ее окружение**.
- Вызывает все деструкторы для объектов, которые имеют статический класс памяти.

void abort()

- Вызывает немедленное завершение прграммы***.
- Обычно вызывает некоторые дополнительные действия, зависящие от реализации****

int atoi(char *)

- Преобразует строку в целое.

long atol(char *)

- Преобразует строку в длинное целое.

double atof(char *)

- Преобразует строку в число с плавающей точкой двойной точности.

---

*Оператор return в функции main() эквивалентен вызову функции exit() с возвращаемым значением, передаваемым в качестве параметра.

**Значения таких кодов и их значения зависят от реализации, хотя нулевое значение, как правило, означает успешное завершение программы, а ненулевое - ошибку.

***Эта функция отличается от exit() тем, что не вызывает деструкторы.

****Например, сохраняет на диск содержимое памяти.

Набор символов ASCII

Ниже приведена таблица символов ASCII, значения которых приведены в десятичной, восьмиричной и шестнадцатиричной системе счисления. Для управляющих символов, то есть для символов с кодами, меньшими 32, приведены также и их мнемонические обозначения (обозначение '^' говорит о том, что данный код получается при нажатии клавиши Ctrl и соответствующей буквы).

Dec	Oct	Hex	Клавиша	Мнемоника	Значение
0	000	00	^@	NUL	Ноль
1	001	01	^A	SOH	Начало заголовка
2	002	02	^B	STX	Начало текста
3	003	03	^C	ETX	Конец текста
4	004	04	^D	EOT	Конец передачи
5	005	05	^E	ENQ	Запрос
6	006	06	^F	ACK	Подтверждение
7	007	07	^G	BEL	Сигнал
8	010	08	^H	BS	Забой
9	011	09	^I	HT	Горизонтальная табуляция
10	012	0A	^J	LF	Перевод строки
11	013	0B	^K	VT	Вертикальная табуляция
12	014	0C	^L	FF	Прогон формата
13	015	0D	^M	CR	Возврат каретки
14	016	0E	^N	SO	Выключить сдвиг
15	017	0F	^O	SI	Включить сдвиг
16	020	10	^P	DLE	Ключ связи данных
17	021	11	^Q	DC1	Управление

					устройством 1
18	022	12	^R	DC2	Управление устройством 2
19	023	13	^S	DC3	Управление устройством 3
20	024	14	^T	DC4	Управление устройством 4
21	025	15	^U	NAK	Отрицательное подтверждение
22	026	16	^V	SYN	Синхронизация
23	027	17	^W	ETB	Конец передаваемого блока
24	030	18	^X	CAN	Отказ
25	031	19	^Y	EM	Конец среды
26	032	1A	^Z	SUB	Замена
27	033	1B	^[	ESC	Ключ
28	034	1C	^\	FS	Разделитель файлов
29	035	1D	^]	GS	Разделитель группы
30	036	1E	^^	RS	Разделитель записей
31	037	1F	^_	US	Разделитель модулей

Dec	Oct	Hex	Клавиша	Dec	Oct	Hex	Клавиша
32	040	20	(пробел)	80	120	50	P
33	041	21	!	81	121	51	Q
34	042	22	"	82	122	52	R
35	043	23	#	83	123	53	S
36	044	24	$	84	124	54	T
37	045	25	%	85	125	55	U
38	046	26	&	86	126	56	V
39	047	27	'	87	127	57	W
40	050	28	(	88	130	58	X
41	051	29	)	89	131	59	Y
42	052	2A	*	90	132	5A	Z

Dec	Oct	Hex	Клавиша	Dec	Oct	Hex	Клавиша
43	053	2B	+	91	133	5B	[
44	054	2C	,	92	134	5C	\
45	055	2D	-	93	135	5D	]
46	056	2E	.	94	136	5E	^
47	057	2F	/	95	137	5F	_
48	060	30	0	96	140	60	`
49	061	31	1	97	141	61	a
50	062	32	2	98	142	62	b
51	063	33	3	99	143	63	c
52	064	34	4	100	144	64	d
53	065	35	5	101	145	65	e
54	066	36	6	102	146	66	f
55	067	37	7	103	147	67	g
56	070	38	8	104	150	68	h
57	071	39	9	105	151	69	i
58	072	3A	:	106	152	6A	j
59	073	3B	;	107	153	6B	k
60	074	3C	<	108	154	6C	l
61	075	3D	=	109	155	6D	m
62	076	3E	>	110	156	6E	n
63	077	3F	?	111	157	6F	o
64	100	40	@	112	160	70	p
65	101	41	A	113	161	71	q
66	102	42	B	114	162	72	r
67	103	43	C	115	163	73	s
68	104	44	D	116	164	74	t
69	105	45	E	117	165	75	u
70	106	46	F	118	166	76	v
71	107	47	G	119	167	77	w
72	110	48	H	120	170	78	x
73	111	49	I	121	171	79	y
74	112	4A	J	122	172	7A	z
75	113	4B	K	123	173	7B	{
76	114	4C	L	124	174	7C	|
77	115	4D	M	125	175	7D	}
78	116	4E	N	126	176	7E	~
79	117	4F	O	127	177	7F	DEL

Содержание

Учебное издание

Пол Дж. Лукас
C++ под рукой

Ответственный за выпуск
В. И. Антоненко
Художественный редактор
С. Н. Сущенко
Технический редактор
Ю. Н. Артеменко
Корректоры
Ю. Н. Артеменко,
С. Н. Сущенко

Сдано в набор 1.06.93. Подписано к печати 8.06.93. Формат 84x108 1/32.
Бумага офсетная. Гарнитура "Таймс". Печать офсетная. Усл. печ. л. 8,38.
Усл. краскоот. 8,80. Уч.-изд. л. 7,59. Тираж 40000 экз. Зак. № 3-207
Цена договорная.

Издательство Научно-исследовательская проектная фирма «ДиаСофт»
Адрес: 252055, Киев-55, а/я 100.
Тел./Факс (044) 441-9766.

Полиграфкомбинат «Украина»
Адрес: 254119, Киев-119, ул. Дегтяревская, 38—44.

Издательская политика фирмы ДиаСофт.

Фирма ДиаСофт планирует выпускать следующие книжные серии:

* **языки программирования;**
* **программирование в среде Windows;**
* **технологии программирования;**
* **объектно-ориентированные библиотеки;**
* **средства автоматизации программирования;**
* **семантические справочники;**
* **диалектика программирования;**
* **искусственный интеллект;**

Дополнительно по каждой из серий мы можем сообщить следующую информацию.

Серия "Языки программирования"

Будет состоять из книг описывающих различные перспективные языки программирования, такие как объектно- ориентированный Паскаль, Си++, Лисп, Пролог и др.

При описании этих языков мы будет стремиться использовать самые новые материалы и излагать их в простой и доступной форме без ущерба качеству.

Серия "Программирование в среде Windows"

Книги этой серии будут обеспечивать возможность программистам с легкостью разобраться в особенностях и перспективах использования оболочки Windows. Мы надеемся, что сможем предоставить полную информацию о всех средствах, позволяющих упростить программирование в среде Windows.

Серия "Технологии программирования"

Книги этой серии будут предоставлять читателям полную информацию о всех перспективных технологиях программирования. Серия будет содержать информацию как о самых новых, так и о только зарождающихся технологиях и тенденциях в программировании. В книгах этой серии будет сделана попытка формализации технологии

программирования и практическая иллюстрация излагаемых аспектов.

Материал, излагаемый в книгах этой серии, будет носить преимущественно теоретический характер, не привязанный к конкретным формам реализации.

Серия "Объектно-ориентированные библиотеки"

Книги серии будут содержать материалы по объектно ориентированным библиотекам, представляющим интерес для профессиональных программистов.

Серия "Средства автоматизации программирования"

Задача данной серии состоит в популяризации различных систем автоматизации программирования, таких как CASE- систем, систем визуального проектирования и других, увеличивающих производительность труда программиста.

Серия "Семантические справочники"

В книгах этой серии будут собраны различные справочные материалы, не привязанные к конкретным системам программирования. В этой серии мы попытаемся излагать материал терминологического характера, а также материал, позволяющий осмыслить то или иное терминологическое понятие и в дальнейшем использовать его в своей работе. Возможны программные реализации излагаемого материала. Коллектив нашей фирмы надеется, что справочники этой серии сыграют большую роль в становлении понятийного аппарата в среде программистов. К составлению данных справочников будет привлекаться целая группа программистов, имеющих большой опыт работы в различных областях программирования. Источниками информации будет как зарубежная литература, так и отечественная, к тому же многие понятия будут предварительно обсуждаться в массовом порядке на различных программистких собраниях.

Справочники предполагается издавать с определенной регулярностью по мере накопления нового материала. Предполагается также организовать широкомасштабную компанию по сбору предложений и дополнений к данным

справочникам. Можно предвидеть уже на данном этапе, что справочникам с течением времени будут требоваться как структурные так и содержательные изменения, поэтому составители данных справочников будут в этом плане минимально консервативны.

Серия "Диалектика программирования"

Книги этой серии включают материал о истории развития программирования, о новых тенденциях и идеях, появляющихся в процессе развития программирования. Также предполагается давать материал о проблемах стоящих перед программированием, будут делаться попытки философского осмысления протекающих здесь процессов. Главная задача этой серии - показать пути развития программирования и переосмыслить достигнутые результаты в этой сфере. Материал носит чисто теоретический характер.

Серия "Искусственный интеллект"

Представляет материал о различных моделях знаний, а также сравнительные характеристики и области применения этих моделей. В серию будет включаться материал о современных экспертных системах и способах их использования и реализации. Будут обсуждаться различные практические и теоретические проблемы, существующие в этой сфере. Предполагается давать теоретичекий материал в виде гипотез, предположений и т.д., касательно идей по реализации, использованию и применению систем искусственного интеллекта.

Глубокоуважаемые читатели!

Поскольку Ваше мнение в вопросе издания книг является главенствующим, коллектив нашей фирмы просит Вас присылать свои замечания и пожелания по поводу нашей издательской политики. Мы обязательно учтем информацию, любезно присланную Вами по адресу: Украина, 252055, г. Киев-55, а/я 100, фирма ДиаСофт.

Помогите нам сделать наши книги лучше!